ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta strojní – Ústav přístrojové a řídicí techniky

DIPLOMOVÁ PRÁCE

Vektorové řízení synchronního motoru s permanentními magnety s vývojovou deskou PYNQ-Z2

Vector control of permanent magnet synchronous motor implemented on development board PYNQ-Z2

Tomáš Buriánek

2024

Prohlašuji, že jsem tuto práci vypracoval samostatně s použitím literárních zdrojů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů.

Datum: ....

Podpis

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Buriánek** Jméno: **Tomáš** Osobní číslo: **482391**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vektorové řízení synchronního motoru s permanentními magnety s vývojovou deskou PYNQ-Z2

Název diplomové práce anglicky:

Vector control of permanent magnet synchronous motor implemented on development board PYNQ-Z2

Pokyny pro vypracování:

1) Proveďte rešerši se zaměřením na aplikace programovatelných hradlových polí (FPGA, Field Programmable Gate Array) v průmyslu a výzkumu. Soustřeďte se na jejich použití ve spojení s výkonovou elektronikou a řízením elektrických pohonů. Popište strukturu a princip funkce FPGA.
2) Seznamte se s metodami programování hradlových polí. Zaměřte se na vývojová prostředí Vivado a Simulink.
3) Realizujte a naprogramujte vektorové řízení synchronního motoru s permanentními magnety (PMSM) na vývojové desce PYNQ-Z2. Vytvořte vhodnou konzoli pro komunikaci s vývojovou deskou tak, aby bylo možné posílat příkazy pro řízení motoru po komunikační lince a zároveň monitorovat základní údaje o chování motoru (rychlost, proudy, apod.).
4) Navržené řešení otestujte na laboratorním PMSM. Shrňte dosažené výsledky s vývojem aplikace a vektorovým řízením za pomoci PYNQ-Z2.

Seznam doporučené literatury:

[1] BUSO, Simone a Paolo MATTAVELLI. Digital Control in Power Electronics. San Rafael, CA: Morgan & Claypool, 2006.
[2] KRISHNAN, Ramu. Permanent Magnet Synchronous and Brushless DC Motor Drives. Boca Raton, FL: CRC Press, 2010. ISBN 9781315221489.
[3] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan and M. W. Naouar, "FPGAs in Industrial Control Applications," in IEEE Transactions on Industrial Informatics, vol. 7, no. 2, pp. 224-243, May 2011.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Zdeněk Novák, Ph.D. U12110.1

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **27.10.2023** Termín odevzdání diplomové práce: **19.01.2024**

Platnost zadání diplomové práce: _____

Ing. Zdeněk Novák, Ph.D.	prof. Ing. Tomáš Vyhlídal, Ph.D.	doc. Ing. Miroslav Španiel, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce, panu Ing. Zdeňku Novákovi, Ph.D., za zapůjčení použitého hardwaru, poskytnuté konzultace a vstřícný přístup.

Abstrakt

Tato diplomová práce se zaměřuje na vytvoření vektorového řízení synchronního motoru s permanentními magnety (PMSM) Teknic M-2310P-LN-04K na vývojové desce PYNQ-Z2. Práce také popisuje princip fungování programovatelných hradlových polí (FPGA) a způsob jejich programování. Při tom se zaměřuje na vývojová prostředí Vivado a Simulink. V praktické části je realizováno vektorové řízení s regulací rychlosti a jeho uživatelské rozhraní. Uživatel může nastavovat veškeré parametry regulace a průběžně sledovat a zaznamenávat průběhy řízených a doplňkových veličin. Řízení bylo úspěšně otestováno s dobrými výsledky. Výpočetní čas je roven 325 ns, při započítání doby měření se jedná celkem o 3 925 ns. Nejvyšší nastavitelná frekvence PWM je 100 kHz, zároveň je možné zaznamenávat vzorky zvolených veličin z každé periody PWM.

Klíčová slova: vektorové řízení, synchronní motor s permanentními magnety, PMSM, programovatelná hradlová pole, FPGA, pulzně šířková modulace napětí, PWM, prostorově vektorová modulace napětí, SVM, systém na čipu, SoC, PYNQ-Z2

Abstract

This thesis focuses on the development of a vector control of a Teknic M-2310P-LN-04K permanent magnet synchronous motor (PMSM) on the PYNQ-Z2 development board. The thesis also describes the principle of operation of field programmable gate arrays (FPGAs) and how to program them. In doing so, it focuses on the Vivado and Simulink development environments. In the practical part, the vector control with speed control and its user interface are implemented. The user can set all control parameters and continuously monitor and record the waveforms of the controlled and additional variables. The control has been successfully tested with good results. The computation time is equal to 325 ns, with a total of 3 925 ns when the measurement time is included. The highest adjustable PWM frequency is 100 kHz, at the same time it is possible to record samples of selected quantities from each PWM period.

Keywords vector control, permanent magnet synchronous motor, PMSM, field programmable gate array, FPGA, pulse width modulation, PWM, space vector modulation, SVM, system on chip, SoC, PYNQ-Z2

Obsah

1 Úvod

Elektromotory jsou v současné době významnou součástí průmyslových aplikací. S rostoucími nároky na přesnost jejich řízení, efektivitu a spotřebu elektrické energie rostou i nároky na výkon výpočetní techniky. V těch nejnáročnějších aplikacích proto nemusí být dostačující klasické mikrokontroléry spoléhající se na sekvenční vykonávání instrukcí. V takových případech je možné využít programovatelná hradlová pole (FPGA, ***F****ield-****p****rogrammable* ***g****ate* ***a****rray*), případně jejich kombinaci s klasickými mikrokontroléry.

Tato práce se zaměřuje na vytvoření vektorového řízení motoru s permanentními magnety (PMSM, ***P****ermanent-****m****agnet* ***s****ynchronous* ***m****otor*) Teknic M-2310P-LN-04K s využitím posledního zmíněného přístupu. K tomu je využita vývojová deska PYNQ-Z2, která je osazena čipem kombinujícím výkonný dvoujádrový mikrokontrolér s FPGA. Samotný řídicí algoritmus je celý realizován v rámci FPGA, klasická část procesoru zajišťuje komunikaci s uživatelským rozhraním realizovaným mimo vývojovou desku.

2 Využití a princip programovatelných hradlových polí

2.1 Zasazení FPGA do kontextu výpočetní techniky ve vestavěných systémech

Pro pochopení výhod využití programovatelných hradlových polí v průmyslových aplikacích je nejdříve nutné uvést jejich základní princip. Veškerá digitální výpočetní technika je tvořena integrovanými obvody. Podle účelu použití a univerzálnosti lze tyto obvody rozdělit do různých kategorií, mezi něž patří:

- Zákaznické integrované obvody (ASIC, ***a**pplication-**s**pecific **i**ntegrated **c**ircuit*)
- Programovatelná hradlová pole (FPGA, ***f**ield-**p**rogrammable **g**ate **a**rray*)
- Mikroprocesory

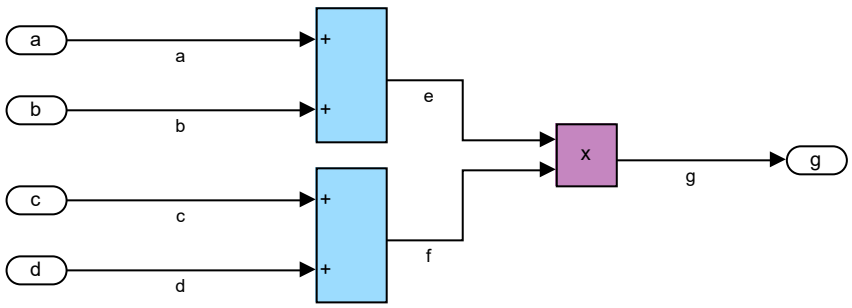
První zmíněnou kategorií jsou zákaznické integrované obvody. Jedná se o jednoúčelové integrované obvody navržené pro konkrétní aplikaci bez jakékoli flexibility nebo možnosti následného přeprogramování. Díky možnosti optimalizace do posledního detailu sice nabízejí nejvyšší výpočetní rychlost, počáteční investice je však velmi vysoká. Jejich využití je proto vhodné ve velkých sériích, případně v těch nejextrémnějších aplikacích, kde žádná jiná možnost není dostatečná.

Vzhledem k tomu, že každý výpočet má svou vlastní část obvodu, je možné provádět výpočty paralelně v čase kdekoli to logika výpočtů dovolí. Jednoduchým příkladem takového výpočtu může být kombinační funkce

$$g = (a + b) \cdot (c + d) \tag{2.1}$$

graficky znázorněná na obrázku 2.1. Výpočet součtu $c + d$ zde není závislý na výsledku součtu $a + b$ a naopak, logika výrazu tedy nebrání souběžnému výpočtu. Součin naopak musí počkat na výsledky obou dílčích výpočtů.

Opakem z hlediska flexibility jsou mikroprocesory. Jedná se o vysoce univerzální sekvenční integrované obvody provádějící výpočty na základě strojových instrukcí.



Obr. 2.1: Grafické znázornění příkladu výpočtu, který lze z části provádět paralelně

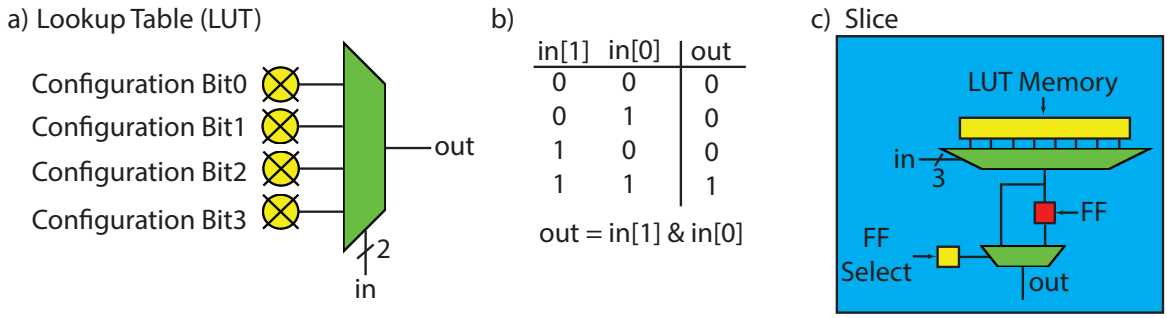
Ke svému fungování potřebují operační paměť (může být integrována přímo v čipu – mikrokontroléry), ve které je uložen program a zpracovávané hodnoty (data). Výhodou je, že lze vykonávaný program snadno a rychle měnit. Nevýhodou je, že v jednom okamžiku je možné provádět jen jednu instrukci na každém jádře (běžné mikrokontroléry obsahují jen jedno jádro). V případě výrazu (2.1) by tak mikroprocesor musel provést součty postupně. Navíc je zdržován dalšími nezbytnými operacemi, jako je načítání a ukládání dat z a zpět do paměti. Další výhodou mikroprocesorů je jejich cenová přijatelnost i v případě potřeby byť jediného kusu, neboť jsou vyráběny ve velkých sériích a díky své univerzálnosti je tak každý model použitelný v široké škále konkrétních aplikací.

Kompromisem mezi těmito variantami jsou programovatelná hradlová pole. Jedná se o pole programovatelných logických a paměťových bloků navržených výrobcem, které jsou vzájemně propojeny pomocí programovatelných spojů. V porovnání s ASIC jsou sice méně efektivní, plně si však zachovávají jejich možnosti paralelních výpočtů. Každá část výpočtů má totiž opět svou vlastní část hardwaru. V současné době navíc výrobci přidávají kromě základních logických a paměťových prvků i specializované prvky zaměřené na konkrétní způsoby použití, čímž přibližují efektivitu FPGA k ASIC obvodům. Více o struktuře a principu FPGA je popsáno v kapitole 2.2. V porovnání s mikroprocesory je však programování stále časově náročnější a vyžaduje složitější verifikační proces. Nelze například zastavit vykonávání programu v určitém bodě a zobrazit aktuální hodnoty, tak jak je dnes běžné u klasických mikroprocesorů. Díky své univerzálnosti jsou však stejně jako mikroprocesory vyráběny v relativně velkých sériích využitelných širokou škálou zákazníků, což je činí výrazně cenově dostupnějšími než ASIC obvody. [1; 2; 3]

2.2 Princip FPGA

Základním typem programovatelných logických prvků jsou takzvané **L**ook**u**p **T**ables (LUT). Jedná se o prvky schopné provádět logické operace formou pravdivostních tabulek. Ke každé kombinaci bitů na vstupu je přiřazen binární výsledek pomocí konfiguračního bitu. Jejich velikost je dána výrobcem, minimální smysluplná délka vstupu je 2 bity, v praxi se však v dnešní době používají větší tabulky (obvykle 4-6 bitů [3], v čipu ZYNQ XC7Z020-1CLG400C SoC použitém v této práci je to 6 bitů [4]). Příklad takové minimální LUT je znázorněn na obrázku 2.2 a). Na obrázku 2.2 b) je pak zobrazena konfigurace pro logický součin vstupních bitů.

Základním typem paměťových prvků jsou klopné obvody (FF, ***f**lip-**f**lop*). Stejně jako klasické mikroprocesory potřebují i programovatelná hradlová pole zdroj hodinového signálu. Bez něho by nebylo možné provádět sekvenční logické funkce a výpočty, pouze čistě kombinační. Tento hodinový signál je využíván jako zdroj pro inicializaci změny

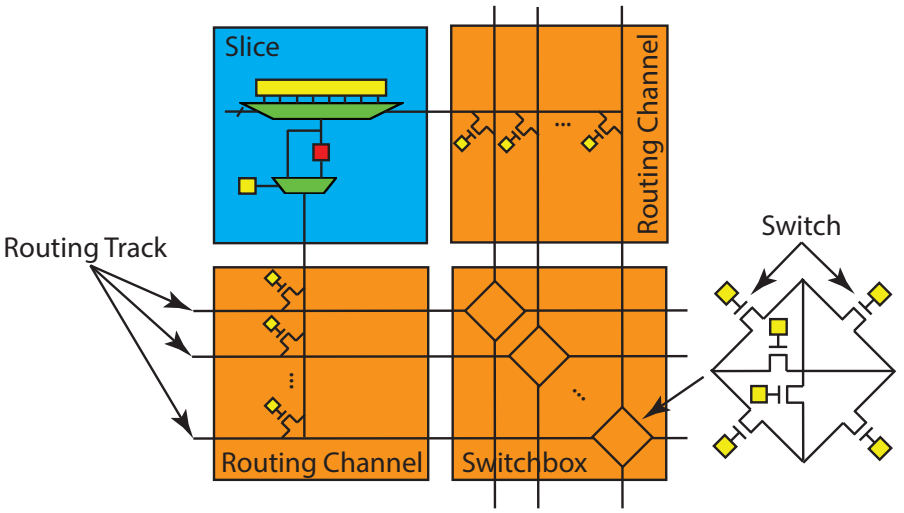


Obr. 2.2: a) Dvoubitová LUT; b) Příklad konfigurace dvoubitové LUT - logický součin; c) Příklad uspořádání programovatelného logického bloku [3]

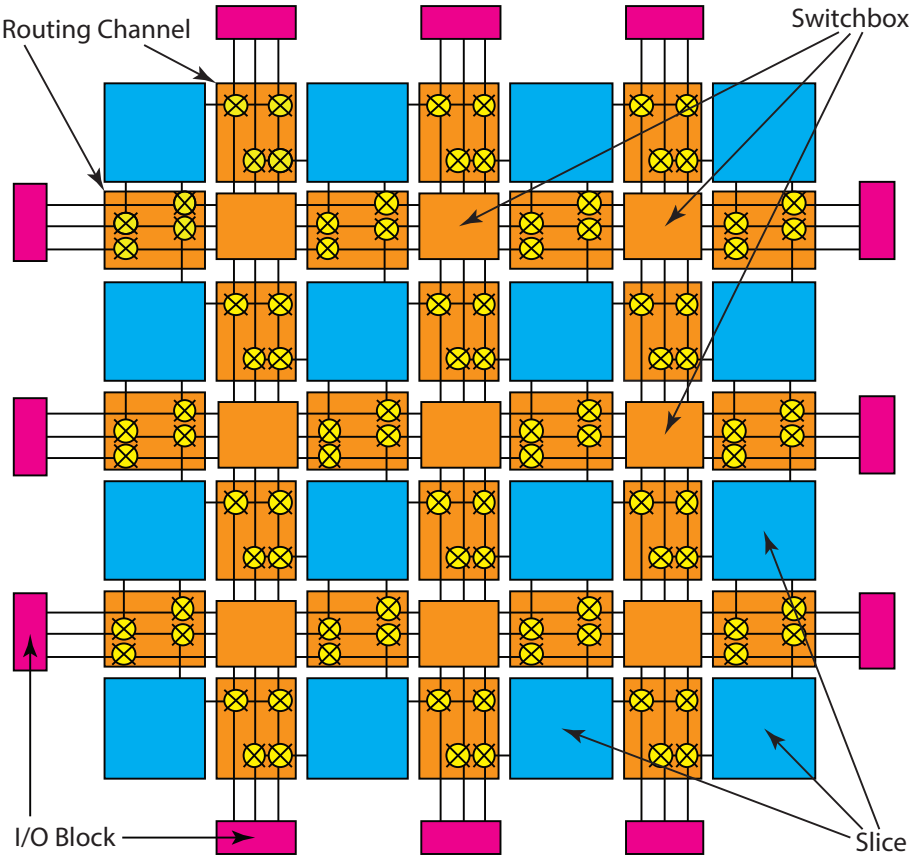
stavu klopných obvodů, kdy dojde k promítnutí vstupní hodnoty na výstup a jejímu držení na výstupu až do příští inicializace. Zdroj pro inicializaci funkce jednotlivých klopných obvodů (nebo jejich malých skupin) je volitelný, v rámci FPGA tak může být zavedeno více různých hodinových signálů, a to jak synchronních (odvozených ze stejného zdroje), tak i asynchronních.

Aby bylo možné ze zmíněných základních prvků vytvořit komplikovanější výpočty a algoritmy, je nutné mít možnost je vzájemně pospojovat. Prvky jsou nejdříve lokálně uspořádány do menších polí, označovaných jako programovatelné logické bloky, nebo také *slice* (existují i jiná označení, slice je však používáno výrobcem hardwaru, který byl použit v této práci). Každý programovatelný logický blok obsahuje určité množství vzájemně pospojovaných základních prvků. Příklad jednoduchého uspořádání logického bloku je zobrazen na obrázku 2.2 c). Tento logický blok obsahuje jednu 3bitovou LUT a jeden klopný obvod, který umožňuje uložení výsledku kombinační funkce realizované pomocí LUT. Z obrázku je patrné, že k uložení hodnoty do klopného obvodu (v tomto konkrétním uspořádání) dochází pokaždé, konfiguračním bitem označeným *FF Select* je však provedena volba, zda bude výsledek logické operace promítnut na výstup okamžitě, nebo až v dalším cyklu. V případě hardwaru použitého v této práci se na každém logickém bloku nachází 4 LUT, 8 FF a několik dalších prvků, všechny prvky na stejném bloku mají navíc společný zdroj hodinového signálu.

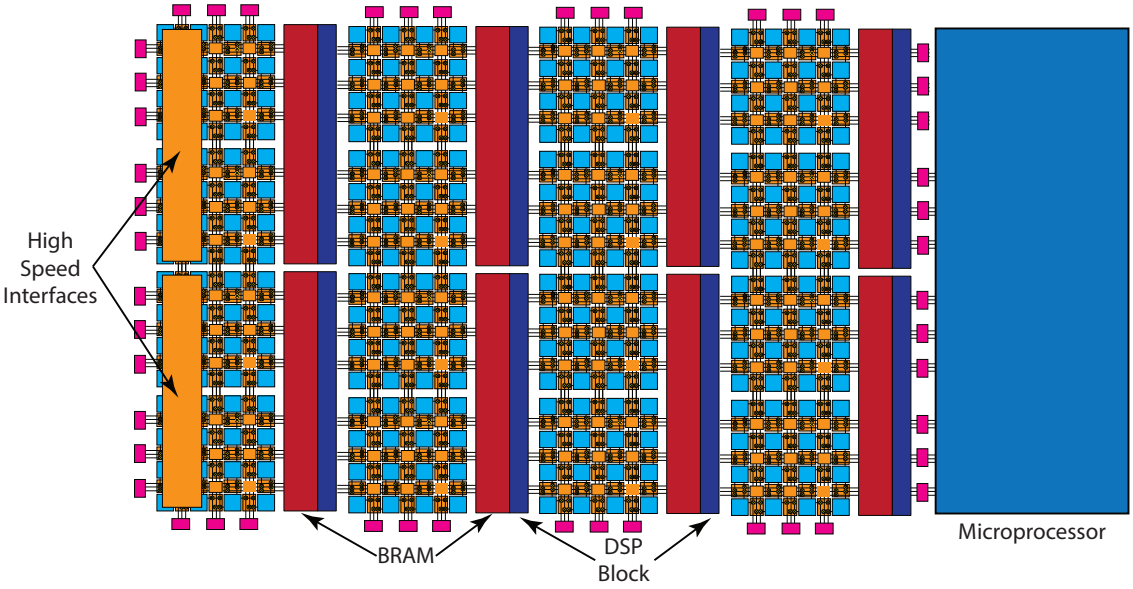
Kromě lokálních spojů na jednotlivých logických blocích je zapotřebí zajistit propojení prvků i mezi jednotlivými bloky. K tomu slouží systém trasování, znázorněný na obrázku 2.3. Každý vstup a výstup programovatelného logického bloku je připojitelný na několik tras (routing track) pomocí tranzistorů na trasovacím kanále (routing channel). Každý tranzistor představuje z hlediska programování jeden konfigurační bit a umožňuje připojit vstup nebo výstup na konkrétní trasu. Trasy jsou mezi jednotlivými kanály propojeny pomocí matic spínačů (switchbox). V každé matici lze pomocí jednotlivých, samostatně konfigurovatelných spínačů (switch) propojit trasy



Obr. 2.3: Znázornění principu propojení vstupů a výstupů jednotlivých logických bloků [3]



Obr. 2.4: Příklad 2D struktury programovatelných hradlových polí složených ze základních programovatelných logických bloků, systému trasování a vstupně-výstupních bloků [3]



Obr. 2.5: Příklad 2D struktury programovatelných hradlových polí obsahujících specializované prvky, propojených s mikroprocesorem [3]

mezi několika kanály, pro zajištění co největší variability ideálně každou s každou. Konkrétní podoba propojení však závisí na výrobci hardwaru, v příkladu na obrázku 2.3 je využita 2D struktura. Stejná 2D struktura je využita i na obrázku 2.4, kde každá matice spínačů vzájemně propojuje trasy ze 4 kanálů.

Provádění matematických výpočtů pomocí obvodů vytvořených z pravdivostních tabulek a klopných obvodů je sice možné, nejedná se však o efektivní řešení. Výrobci hardwaru proto přidávají na programovatelné logické bloky i neprogramovatelné specializované prvky, jako jsou sčítačky, které slouží k předem danému účelu. Zároveň přidávají do FPGA samostatně i větší specializované bloky, jako jsou DSP (***d**igital **s**ignal **p**rocessing*) bloky zaměřené na aritmetické operace a logické operace s celými slovy (ve smyslu datových proměnných, například 16bitová celá čísla), nebo BRAM (***b**lock **r**andom-**a**ccess **m**emory*) pro ukládání většího množství dat (obvykle řádově desítky kilobitů v jedné BRAM). Jednotlivé BRAM je možné v závislosti na konkrétní implementaci výrobce konfigurovat pro různé chování, což zahrnuje například různé délky slova, řazení jednotlivých bloků do série pro vytvoření většího celku, nebo přepnutí do FIFO (fronta, ***f**irst **i**n, **f**irs **o**ut*) režimu. V případě použitého ZYNQ XC7Z020 má každý blok RAM kapacitu 36 kilobitů s délkou slova nastavitelnou v mocninách 2 (počínaje $2^0 = 1$) až do hodnoty 72 bitů.

Aby měla programovatelná hradlová pole vůbec smysl, musí mít možnost interagovat s okolním světem. K tomu slouží v první řadě digitální vstupně-výstupní bloky, které zajišťují převod napěťových úrovní mezi FPGA a externími zařízeními, jako jsou například senzory, akční členy v případě regulačních aplikací nebo velkokapacitní

externí paměť pro práci s velkým množstvím dat. Výrobci však do FPGA přidávají stále více specializovaných bloků určených pro interakci s okolím. Často se tak lze setkat se zabudovanými analogově-digitálními (A/D) převodníky pro čtení analogových signálů nebo bloky pro standardizovanou vysokorychlostní komunikaci, jako například ethernet pro přístup k internetu a HDMI pro přenos grafického signálu. V neposlední řadě se lze setkat s FPGA, které jsou integrované v jednom čipu s klasickým mikroprocesorem (potenciálně i více FPGA a mikroprocesorů v jednom čipu). V takovém případě obvykle obsahují rozhraní optimalizované pro efektivní spolupráci s daným mikroprocesorem a dalšími komponenty (paměť využívaná procesorem apod.) Příklad struktury jednoduchého FPGA s integrovaným mikroprocesorem, DSP bloky, bloky RAM a vysokorychlostním rozhraním je znázorněn na obrázku 2.5. Stejně jako na obrázku 2.4 je zde však zobrazeno jen velmi malé množství logických bloků. Reálná FPGA mají desítky tisíc až miliony programovatelných logických bloků, v současné době nejvíce logických bloků mezi čipy vyráběnými společností Advanced Micro Devices, Inc. (AMD) má čip Versal Premium VP1902 [5][1], a to přes 18,5 milionu. [3; 4; 8; 9; 10; 1]

2.3 Využití FPGA v průmyslových aplikacích

2.3.1 Přechod z externích komponent na FPGA

Existuje mnoho důvodů, proč při návrhu vestavěných systémů využít programovatelná hradlová pole. Jedním z nich je dlouhodobá dostupnost navržených algoritmů. Dnešní moderní systémy a zařízení jsou často výsledkem velmi dlouhého vývoje, kdy společnosti vycházejí ze svých předešlých produktů a neustále je vylepšují. Často tak ale dochází k tomu, že původní elektronické systémy a specializované výpočetní obvody, na kterých je celý návrh závislý, nejsou na trhu již nadále k dispozici [1]. V takovém případě je nutné najít náhradu, což s sebou nese další komplikace, neboť je nutné celý návrh upravit tak, aby byl kompatibilní s novými komponenty. To však může vést ke znehodnocení optimalizací, které byly provedeny s ohledem na specifikace původních systémů. S neustále rostoucími nároky na optimalizaci z důvodu vyšších nároků na přesnost, nižší spotřebu energie apod. je tento problém stále výraznější. Pokud je však algoritmus realizován čistě pomocí FPGA, mělo by být možné ho jednoduše přesunout na novější verzi hardwaru, aniž by přitom došlo ke změně jeho vlastností. I zde je sice úskalí, že pokud byl algoritmus realizován na konkrétním hardwaru

[1] 14.02.2022 dokončila společnost Advanced Micro Devices, Inc. akvizici společnosti Xilinx, Inc. [6; 7], jejíž produkty jsou využívány v této práci. V době psaní této práce je technická dokumentace dostupná stále na původní doméně `xilinx.com`. Dokumenty, které byly od té doby aktualizovány, již obsahují název společnosti AMD, starší dokumenty ale stále obsahují původní společnost Xilinx. V souladu s citačními zvyklostmi je v seznamu použité literatury uváděna vždy společnost uvedená v daném dokumentu.

pomocí specializovaných prvků, které nejsou v FPGA moc běžné, může na jiném hardwaru selhat časování. FPGA se však v současné době rychle vyvíjejí a množství specializovaných prvků neustále roste. Lze tedy předpokládat, že v průběhu životního cyklu konkrétního hardwaru přibudou na trhu nové varianty hardwaru, které budou schopné vyhodnocovat algoritmus bez nutnosti jeho úprav.

Kromě dlouhodobé dostupnosti jsou dílčí subsystémy vytvořené v FPGA také velmi dobře integrovatelné do navrhovaného systému, neboť se jedná o čistě softwarové řešení a vývojáři tak nemusí řešit hardwarovou integraci. Stále však mají vyhrazenou svou vlastní část hardwaru a může tak docházet k jejich vyhodnocování nezávisle na zbytku systému (ve smyslu nežádoucího ovlivňování, jako je například přerušení určité operace ve prospěch jiné při použití klasických mikroprocesorů, kde je tak latence závislá na ostatních částech výsledného algoritmu). V případě FPGA nemusí mít vývojář určitého subsystému ponětí o zbytku algoritmu, aby byl schopný zajistit správné fungování v reálném čase (samozřejmě s výjimkou závislostí logiky algoritmu, který navrhuje). To otevírá možnosti pro spolupráci. Společnosti, které nabízely na trhu určitá řešení jako samostatný hardware (což zahrnuje elektronické systémy a výpočetní obvody probírané v předchozím odstavci v souvislosti s jejich životním cyklem), mohou nabízet tato řešení v podobě tzv. duševního vlastnictví (IP, ***i****ntellectual* ***p****roperty*). Jedná se o specifikace, na základě kterých syntetizační program vytvoří vlastní implementaci obsaženého algoritmu, a to bez nutnosti zásahu zákazníka (uživatele IP – vývojáře systému, který IP využívá), nebo s možností nastavení určitých parametrů. Příkladem obecných IP mohou být jádra zajišťující běžně používané funkce v jejich optimalizované podobě, jako jsou DSP výpočty, rozhraní pro standardizované příslušenství, správa paměti, ale také odlehčené mikroprocesory realizované přímo v rámci FPGA. Specializovaná IP mohou zajišťovat například vyhodnocování senzorů, jako je kvadratický inkrementální enkodér elektromotoru, nebo obsahovat celé řídicí algoritmy, případně jejich části. V oblasti výkonové elektroniky a elektromotorů by to mohly být například výpočty zajišťující specifické modulace PWM signálu pro vytvoření požadovaných napětí na jednotlivých fázích motoru.

V návaznosti na první odstavec této podkapitoly je ještě vhodné uvést další potenciální výhodu přechodu z externích komponent na FPGA. Vzhledem k jejich programovatelné povaze lze ve spolupráci s vývojáři daného IP provádět dodatečné úpravy algoritmu pro potřeby konkrétní aplikace. Druhou možností je samozřejmě začít vyvíjet vlastní algoritmus.

2.3.2 Využití vysokého výpočetního výkonu FPGA

Další důvody pro využití programovatelných hradlových polí souvisí s jejich vysokým výkonem. V [11] a [12] bylo popsáno několik možných využití FPGA v oblastech

řízení elektromotorů a výkonové elektronice. Autoři rozdělili případy do dvou základních skupin podle spínací frekvence s pomyslnou hranicí 100 kHz. Rozdělení vychází z toho, že v dnešní době jsou v oblasti digitálně řízených elektrických systémů dvě základní omezení související s dynamikou řízení. Prvním omezením je doba zpracování regulační smyčky, což zahrnuje měření pomocí A/D převodníků, výpočetní čas a (pseudo)digitálně-analogový převod pomocí pulzně šířkové modulace (PWM, ***p**ulse-**w**idth **m**odulation*). Druhým omezením je spínací frekvence výkonových tranzistorů z důvodu spínacích ztrát. Toto omezení je také v dnešní době vlivem vysokého výpočetního výkonu FPGA výraznější.

Existují však aplikace, kde jsou časové požadavky tak přísné, že hlavním omezením je doba zpracování regulační smyčky. Příkladem takové aplikace jsou spínané zdroje, kde spínací frekvence přesahuje 1 MHz a regulační smyčka musí běžet minimálně na stejné frekvenci. Dalším takovým příkladem mohou být HIL simulace (***h**ardware-**i**n-the-**l**oop*). Autoři [11; 12] v tomto případě odkazují na [13], kde byla vytvořena HIL simulace kompletního systému řízení indukčního motoru včetně modelů střídače, motoru, PWM a vektorového řízení. Simulace střídače byla provedena s konstantním krokem 12,5 ns, což umožnilo simulaci nelineární spínací charakteristiky IGBT tranzistoru včetně spínacích ztrát.

Vysoký výpočetní výkon může být výhodou i v aplikacích, kde není vyžadována vysoká vzorkovací frekvence, typicky z důvodu omezení spínacích ztrát. V takových případech může být doba výpočtů tak krátká, že lze upravit žádaná napětí mnohokrát během stejné periody PWM (tzv. *multi-sampled PWM*) a snížit tak latenci mezi měřením a úpravou akční veličiny, což se příznivě projeví na vlastnostech řízení, viz [14]. Samozřejmě to znamená zvolit vhodný způsob měření a jeho filtrace, neboť hladký průběh proudu na motoru je deformován modulací napětí. Příklad konkrétních čísel lze nalézt v [15]. Zde bylo navrženo klasické vektorové řízení (s použitím PI regulátorů) AC motoru se vzorkovací frekvencí 200 kHz (perioda 5 µs) a frekvencí PWM 1 kHz a 3 kHz. Samotný výpočet proudové regulace v tomto případě zabral 0,88 µs a výpočet včetně A/D konverze 3,28 µs. Také zde bylo navrženo prediktivní řízení synchronního motoru s vinutým rotorem. V tomto případě byla vzorkovací frekvence sice nastavena jen na 10 kHz (100 µs), protože docházelo k měření pouze na začátku každé periody, vzhledem k technice modulace a době zpracování pouze 4,52 µs (včetně A/D konverze, samotný výpočet trval 2,12 µs) však byly vypočítané hodnoty promítnuty do modulace hned po dokončení výpočtu.

Z uvedených hodnot je patrné, že v případě nízké vzorkovací frekvence zbývá v rámci každé periody velké množství výpočetního času. Díky tomu mohou být pomocí FPGA implementovány výrazně komplexnější metody řízení v porovnání s klasickými mikrokontroléry. Je možné vytvářet sofistikovanější regulátory a pozorovatele stavu

nebo třeba synchronizované řízení více motorů současně. Stejně tak je možná synchronizace měření s modulací napětí, což může vést k možnosti vynechat filtraci měřených signálů, nebo naopak měřit více vzorků, než je zapotřebí (tzv. *oversampling*), a v kombinaci s možností vytvářet výpočetně náročnější digitální filtry tak zpřesnit měření. Nakonec je možné přidávat další funkce, jako je detekce poškození motoru v reálném čase, prováděná na základě sledování a vyhodnocování chování motoru. [11; 12]

3 Metody programování hradlových polí

Při programování FPGA programátor nenavrhuje svůj algoritmus přímo pomocí elementárních prvků popsaných v kapitole 2.2 a nekonfiguruje přímo jednotlivé konfigurační bity. Místo toho popisuje chování algoritmu na vyšší úrovni v nějakém vhodném jazyce nebo grafickém prostředí. Jeho návrh je pak specializovanými nástroji převeden na konkrétní implementaci v rámci konkrétního FPGA. Tento postup má několik kroků, přesný počet se liší v závislosti na konkrétním způsobu, jakým byl algoritmus definován.

Základním způsobem programování FPGA je návrh na úrovni RTL (***r**egister **t**ransfer **l**evel*). Jedná se o úroveň abstrakce, na které jsou definovány jednotlivé registry a kombinační logika mezi nimi. Pojem registr má z hlediska návrhu stejný význam jako klopné obvody – po dobu celého cyklu udržuje hodnotu, která mu byla přiřazena v předchozím cyklu, čímž umožňuje sekvenční chování. Hodnota přiřazená do registru je definována vždy stejnou kombinační funkcí hodnot dalších registrů (potenciálně i sebe). Oproti klopným obvodům však může registr obsahovat více než jeden bit a reprezentovat tak celou hodnotu.

K vytváření návrhů na RTL úrovni se využívají jazyky spadající do kategorie HDL (***h**ardware **d**escription **l**anguage* neboli jazyk popisující hardware). V současné době nejpoužívanější jsou jazyky Verilog a VHDL. Tyto jazyky původně sloužily k simulování chování digitálních obvodů a jejich dokumentaci. Později se začaly využívat i k návrhu těchto obvodů, a to s pomocí specializovaných programů, které dokáží provést jejich syntézu a vytvořit požadovaný výstup, jako například konfiguraci FPGA. Jedním z takových programů je Vivado, kterému je věnována kapitola 3.1. Vzhledem k jejich původnímu účelu však obsahují HDL jazyky i konstrukty, které nejsou syntetizovatelné. Navíc umožňují popis hardwaru i na jiných úrovních abstrakce než je RTL, což může být matoucí. Ukázka HDL kódu, konkrétně Verilogu, je zobrazena na obrázku 3.1.

Návrh napsaný v HDL kódu (na úrovni RTL) je přenositelný mezi libovolnými modely FPGA, v závislosti na dostupných prvcích a jejich fyzickém provedení však může být časová náročnost algoritmu odlišná, stejně tak množství využitých prvků.

Při vytváření složitějších algoritmů může být návrh na úrovni RTL velmi komplikovaný. Další možností tak je využít vysokoúrovňovou syntézu (HLS, ***h**igh-**l**evel **s**ynthesis*). Jedná se o proces, kdy je program napsaný v klasickém programovacím jazyce nebo jiném nástroji zachycujícím chování algoritmu převeden do HDL kódu na úrovni RTL pomocí specializovaného softwaru. Vzniklý kód je možné následně začlenit do již existujícího RTL projektu. Takto je možné programovat FPGA například s pomocí C, C++, Matlabu nebo dokonce Simulinku.

```
module soucin_souctu(
    input clk,      // zdroj hodinoveho signalu
    input reset,    // resetovaci signal
    input [9:0] a,  // 10 bitu, bez znamenka
    input [9:0] b,  // 10 bitu, bez znamenka
    input [5:0] c,  // 6 bitu, bez znamenka
    input [3:0] d,  // 4 bity, bez znamenka
    output [17:0] g // 18 bitu, bez znamenka
    );

    // inicializace spoju (wire) a promennych (reg)
    wire [10:0] a_cast;
    wire [10:0] b_cast;
    wire [6:0] c_cast;
    wire [6:0] d_cast;
    wire [10:0] e;
    wire [6:0] f;
    wire [17:0] g_temp;
    reg [17:0] g_reg;

    // rozsireni bitove sirky (kombinacni cast)
    assign a_cast  =  {1'b0, a};
    assign b_cast  =  {1'b0, b};
    assign c_cast  =  {1'b0, c};
    assign d_cast  =  {3'b0, d};

    // soucty (kombinacni cast)
    assign e = a_cast + b_cast;
    assign f = c_cast + d_cast;

    // soucin (kombinacni cast)
    assign g_temp = e * f;

    // vytvoreni registru
    always @(posedge clk) begin
        if (reset == 1'b0) begin
            g_reg <= 18'b0;
        end
        else begin
            g_reg <= g_temp;
        end
    end

    // prirazeni hodnoty registru na vystup
    assign g = g_reg;

endmodule
```

Obr. 3.1: Ukázka kódu ve Verilogu. Modul realizuje výpočet výrazu (2.1) pro konkrétní datové typy vstupních dat (nezáporná celá čísla s různými počty bitů), který je následně uložen do registru. Verilog při sčítání ponechá výsledek v bitové šířce širšího z operandů a při přiřazení do spoje/proměnné s jinou šířkou pouze upraví šířku výpočtu. Samotný fakt, že spoje e a f pojmou celé výsledky součtů, proto nezajistí, že nemůže dojít k přetečení. Za tímto účelem je nejdříve nutné rozšířit vždy alespoň jeden operand na potřebný počet bitů. V případě násobení naopak stačí, aby měl potřebnou šířku spoj, do kterého je přiřazováno. Změna hodnoty registru je zde inicializována s každou náběžnou hranou hodinového signálu.

V klasických programovacích jazycích žádné registry (ve významu RTL) nejsou, stejně tak se v základním kódu nevyskytuje souběžnost. HLS proto musí sama odhalit části programu, které je možné vykonávat paralelně, a rozhodnout, kam umístit registry, aby bylo dodrženo časování. K tomu potřebuje od programátora dodatečné informace, jako jsou specifikace cílového hardwaru a cílová frekvence hodinového signálu. Dále je možné programu provádějícímu HLS napovídat, jak má konkrétní části programu implementovat. Například při použití nástroje Vitis HLS, který syntetizuje C nebo C++, je možné napovídat pomocí direktiv `#pragma`. Částečnou výjimkou je Simulink, který díky svému principu fungování umožňuje navrhovat algoritmy i na úrovni RTL, stejně tak jako je v něm obvykle přímo zřetelná souběžnost. Simulinku je věnována kapitola 3.2. [2; 3; 16]

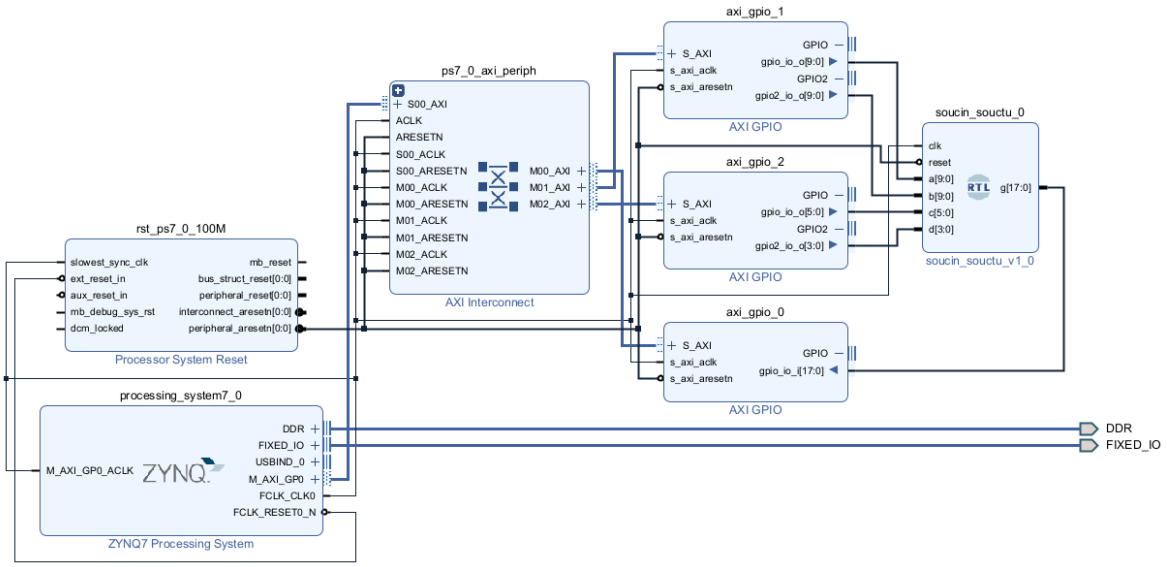
3.1 Vivado

Vivado je software od společnosti AMD (dříve Xilinx) obsahující kompletní sadu nástrojů pro vývoj a implementaci FPGA algoritmů v čipech od AMD (Xilinxu). Umožňuje grafické kombinování RTL modulů a vestavěných či externích IP pomocí blokových schémat, vytváření RTL modulů, vytváření IP, konfiguraci vstupů a výstupů FPGA, simulaci, syntézu a implementaci. Obsahuje i vestavěné moduly IP, které usnadňují komunikaci FPGA s integrovaným mikroprocesorem na čipech řady Zynq 7000 SoC, mezi které patří procesor použitý v praktické části této práce, viz kapitola 6. Tento procesor bude také použit pro všechny ukázky v této kapitole.

Pro práci ve Vivado je nejdříve nutné vytvořit projekt. Při jeho vytváření je nutné zvolit cílovou platformu, lze vybírat ze seznamu podporovaných čipů nebo přímo vývojových desek obsahujících tyto čipy. Do projektu lze také přiřadit externí zdrojové soubory. Základem projektu je blokový design. Ukázka integrace RTL modulu z obrázku 3.1 je zobrazena na obrázku 3.2. Jednotlivé instance IP je možné konfigurovat pomocí grafických dialogů. Zároveň je v blokovém designu možné přidávat vstupy a výstupy a jejich sady, jako jsou na zmíněném příkladu *DDR* a *FIXED_IO*.

Vstupy a výstupy jsou identifikovány podle jejich názvů. Zmíněné dvě sady vstupů a výstupů jsou předkonfigurovány v souborech vývojové desky. Nastavení ostatních vstupů a výstupů se provádí mimo blokový design pomocí tzv. *constraints*[2]. Ty mohou být vytvořeny přímo v příslušném uživatelském rozhraní ve Vivado, nebo importovány do projektu v podobě textových příkazů v textových XDC (***X**ilinx **D**esign **C**onstraints*) souborech. Constraints se obecně používají k poskytnutí dodatečných informací a omezení pro syntézu a implementaci. Kromě fyzického napojení vstupů

[2]Autorovi se nepodařilo dohledat ustálenou českou terminologii, v textu bude proto využíván anglický termín využívaný softwarem Vivado, případně v některých případech přímý překlad. Ten ale není vždy výstižný.



Obr. 3.2: Ukázka blokového RTL schématu ve Vivado. Schéma obsahuje RTL modul z obrázku 3.1 *soucin_souctu_v1_0*, IP pro interakci s mikroprocesorem *ZYNQ7 Processing System*, 3 IP pro čtení a zapisování hodnot pomocí mikroprocesoru *AXI GPIO*, IP propojující 1 master IP a 3 slave IP v rámci AXI komunikace *AXI Interconnect* a IP pro interpretaci resetovacího signálu z mikroprocesoru *Processor System Reset*.

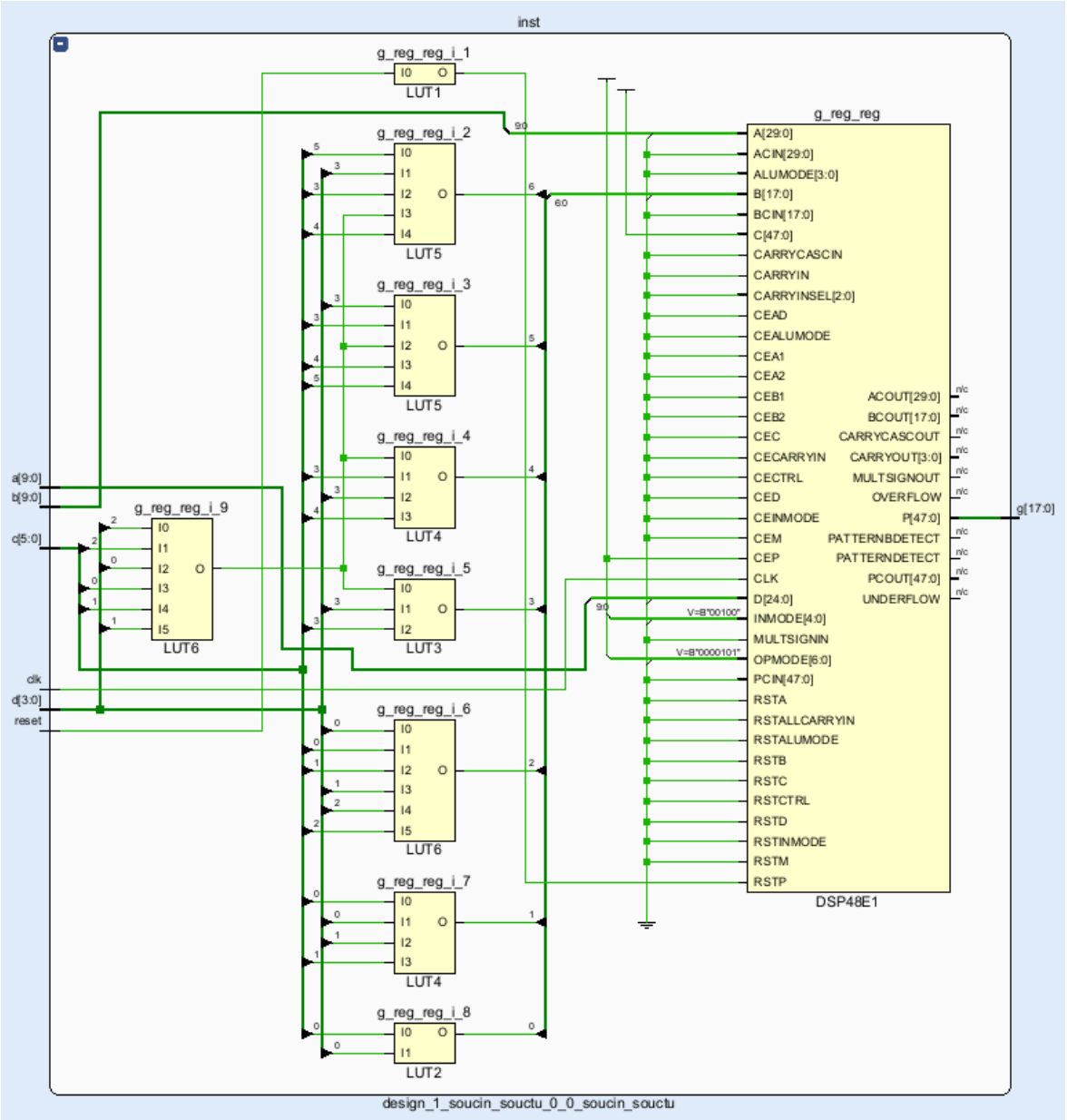
a výstupů se používají například k přidání informací o hodinových signálech, nebo nastavení výjimek z časování. Příkladem constraint může být:

```
create_clock -name clk1 -period 25 [get_ports clk_in1]
```

Tento příkaz Vivado informuje o tom, že vstupní port *clk_in1* je zdrojem hodinového signálu s periodou 25 ns, střídou 50 % a bez fázového posunutí (střída a fázové posunutí nejsou definovány, takže jsou použity výchozí hodnoty). Použití constraints je zdokumentováno v [17].

Vytvořený RTL návrh je nutné syntetizovat. Syntéza převede RTL návrh na úroveň jednotlivých prvků, jejich konfigurací a vzájemných propojení, při čemž bere ohled na prvky dostupné na konkrétním hardwaru. Výsledkem syntézy však ještě není konkrétní rozložení v hardwaru a konfigurace trasování, o to se následně stará implementace. Tyto kroky jsou ve Vivado záměrně odděleny, oba totiž mohou být časově velmi náročné, zvláště u větších algoritmů. Oba je navíc někdy potřeba provádět vícekrát. Z výsledků implementace je nakonec zapotřebí vygenerovat tzv. *bitstream* – soubor, který obsahuje nastavení veškerých konfiguračních bitů FPGA a slouží k naprogramování reálného hardwaru.

Výsledky syntézy a implementace je možné analyzovat. Například je možné zobrazit schéma použitých prvků a jejich propojení nebo počty použitých prvků jednotlivých typů. V případě implementace pak i konkrétní umístění použitých prvků a reálné časy potřebné k vyhodnocení signálů na cestách mezi paměťovými prvky (např. klopnými obvody). V případě neuspokojivého výsledku je možné přidat nebo upravit



Obr. 3.3: Ukázka schématu po dokončení syntézy RTL modulu z obrázku 3.1.

některá omezení (constraints) a spustit daný proces znovu. Například je tak možné požadovat umístění některé LUT získané syntézou na konkrétní logický blok (slice) nebo dokonce na konkrétní fyzickou LUT na daném logickém bloku. Provádění syntézy a implementace je zdokumentováno v [18; 19].

Ukázkový RTL modul z obrázku 3.1 byl při syntéze designu z obrázku 3.2 syntetizován na 9 LUT s různými počty vstupů a 1 DSP blok. Ačkoliv RTL modul obsahuje registr, samostatné FF nebyly vytvořeny, protože jejich funkce je již obsažena v DSP bloku. Ukázka schématu výsledku syntézy tohoto RTL modulu je zobrazena na obrázku 3.3.

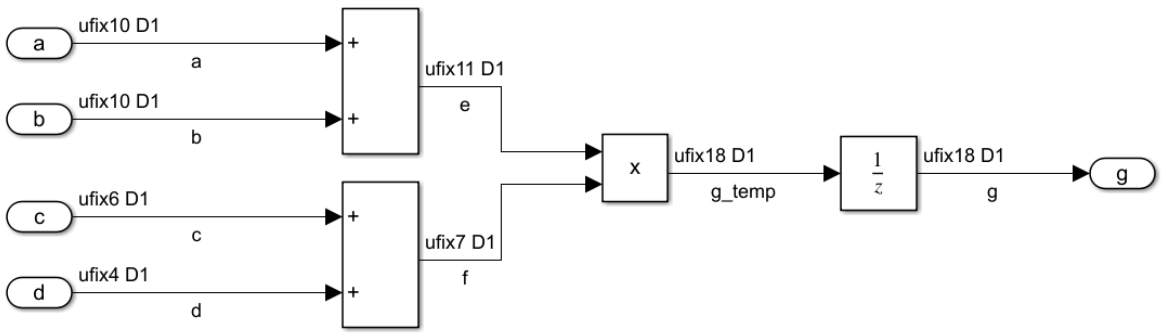
3.2 Simulink

Simulink je grafické prostředí postavené nad Matlabem, programovacím jazykem zaměřeném na technické výpočty, původně určené k modelování a numerickým simulacím dynamických systémů. V průběhu času k němu přibylo mnoho toolboxů, které výrazně rozšiřují jeho oblast použití. V současné době tak umožňuje například generování kódu (C++, PLC, HDL a další) z vytvořených modelů.

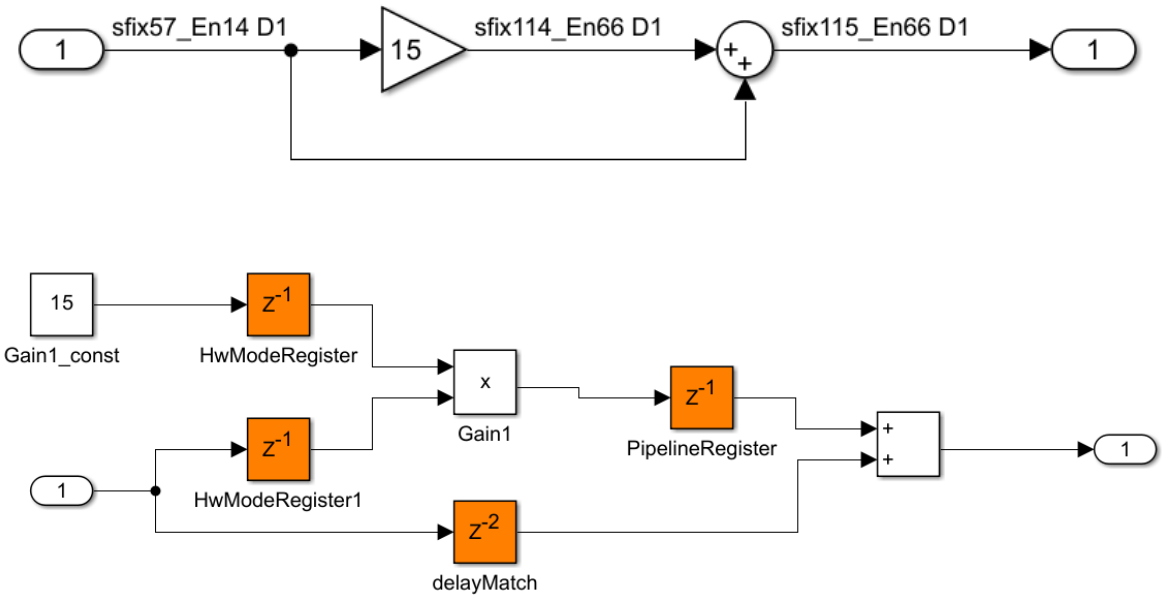
Simulace může probíhat jak ve spojitém, tak i diskrétním čase. V obou případech je model vyhodnocován s určitým časovým krokem, v případě spojitého času jsou numericky řešeny obyčejné diferenciální rovnice v časové doméně. Některé funkce Simulinku vyžadují konkrétní nastavení numerického řešiče. Pro dynamické systémy je výhodná simulace ve spojitém čase s variabilním časovým krokem, pro generování kódu je naopak nutný diskrétní čas s konstantním časovým krokem.

Tvorba modelů probíhá v Simulinku formou blokových schémat. Jednotlivé bloky mohou provádět základní matematické operace, ale i komplexní úlohy. Existuje mnoho předem připravených bloků dostupných v rámci knihoven jednotlivých toolboxů. Navíc je možné do modelu přidávat i funkce napsané přímo v Matlabu.

Způsobů použití Simulinku pro programování FPGA je více. Vzhledem k tomu, že v každém časovém kroku je vyhodnocován celý model, odpovídá simulace v diskrétním čase funkci FPGA. V rámci modelu je navíc možné přidávat bloky zpoždění, které vstupní hodnotu promítnou na výstup až v dalším časovém kroku, což principiálně odpovídá registrům v RTL návrhu. Simulink tak lze použít jednoduše jako grafický editor pro tvorbu RTL návrhů. Příklad subsystému, který provede výpočet výrazu (2.1), je zobrazen na obrázku 3.4. Druhou možností je registry vynechat a nechat Simulink, aby je doplnil sám. Příklad tohoto přístupu je zobrazen na obrázku 3.5. Tyto přístupy je navíc možné kombinovat, v Simulinku je totiž možné vytvářet subsystémy, kterým je následně možné nastavit odlišné chování pro generování HDL kódu.



Obr. 3.4: Ukázka blokového návrhu v Simulinku. Model provádí výpočet výrazu (2.1) a HDL kód z něj vygenerovaný principiálně odpovídá HDL kódu na obrázku 3.1, ačkoli se od něj mírně liší.



Obr. 3.5: Ukázka automatického doplnění registrů v Simulinku. Na prvním obrázku (nahoře) je zobrazen navržený model, na druhém (dole) je model vygenerovaný Simulinkem při generování HDL kódu, který odpovídá vygenerovanému kódu. Oranžově jsou zde automaticky zvýrazněna doplněná zpoždění (registry), horní index u z označuje, o kolik časových vzorků je výsledek posunut.

Dalším důležitým faktem je, že v Simulinku je možné vytvářet vlastní datové typy s pevnou desetinnou čárkou. To totiž opět odpovídá způsobu, jakým se FPGA programují. Simulink navíc umí sám zvolit vhodný datový formát výsledku při provádění matematických operací, stejně tak jako umožňuje uživateli tento formát zvolit. Na obrázcích 3.4 a 3.5 je vidět datový formát zobrazený u jednotlivých signálů ve formátu

$$\mathrm{A}\mathit{fix}\mathrm{B}\_\mathit{En}\mathrm{C}$$

kde A značí čísla se znaménkem (s) nebo bez znaménka (u), B udává celkovou bitovou délku a C polohu desetinné čárky vyjádřenou počtem bitů za ní (C může být v případě potřeby i větší než B, například *ufix10_En15* může reprezentovat pouze nezáporná čísla od nuly do 2^{-5} s rozlišením 2^{-15}). Na obou obrázcích jsou datové typy průběžných a výsledných signálů zvoleny Simulinkem tak, aby nedošlo k přetečení nebo ztrátě přesnosti. Použití datových typů s pevnou desetinnou čárkou v Simulinku zajišťuje toolbox *Fixed-Point Designer*, více informací lze nalézt v příslušné dokumentaci [20].

Simulink umožňuje v případě diskrétního času vytvářet různé části modelu s různými časovými kroky a přechody mezi nimi. To platí i pro modely určené ke generování HDL kódu. V takovém případě je nutné nastavit, zda má mít vygenerovaný RTL modul více vstupů pro hodinové signály, nebo jenom jeden. Ve druhém případě Simulink vytvoří dodatečné řídicí signály pomocí čítače cyklů zdrojového hodinového signálu. Z principu pak musí být všechny časové kroky v modelu a jejich fázová posunutí násobkem periody zdrojových hodin.

Odvozením však z pohledu Vivado nevznikne nový hodinový signál, pouze další logika. Proto Simulink umožňuje automatické vygenerování constraints v podobě XDC souboru, který se importuje do projektu ve Vivado. Zde jsou obsaženy informace o tom, že cesty mezi některými registry mají trvat delší dobu (tzv. vícecyklové cesty, inicializace příslušných klopných obvodů proběhne jen jednou za určitý počet cyklů hodinového signálu). Tento způsob má však pár omezení a v některých případech může dojít k selhání časování, viz [21]. Další informace ke generování HDL kódu lze najít v dokumentaci [22] k toolboxu *HDL Coder*, který generování zajišťuje.

4 Princip vektorového řízení

4.1 Proudová regulace

Vektorové řízení je běžně používaný přístup k řízení třífázových elektromotorů. Jeho základní myšlenkou je využití matematického převodu regulovaných veličin (fázových proudů) z tříosé soustavy statoru do dvouosé soustavy rotoru, ve které je regulace mnohem jednodušší. Časově proměnné harmonické signály (proudy a napětí) jsou totiž převedeny na stejnosměrné a regulace probíhá na konstantní žádanou hodnotu.

Proudy protékající jednotlivými fázemi je možné reprezentovat jako vektory $\vec{i}_a$, $\vec{i}_b$ a $\vec{i}_c$ ležící na příslušných osách A, B a C, reprezentujících jednotlivé fáze statoru. Jejich vektorovým součtem vznikne vektor reprezentující elektromagnetické pole statoru. Tento vektor je následně třeba popsat pomocí pravoúhlého souřadného systému α-β, který je také spojen se statorem. To je možné provést pomocí základních trigonometrických funkcí.

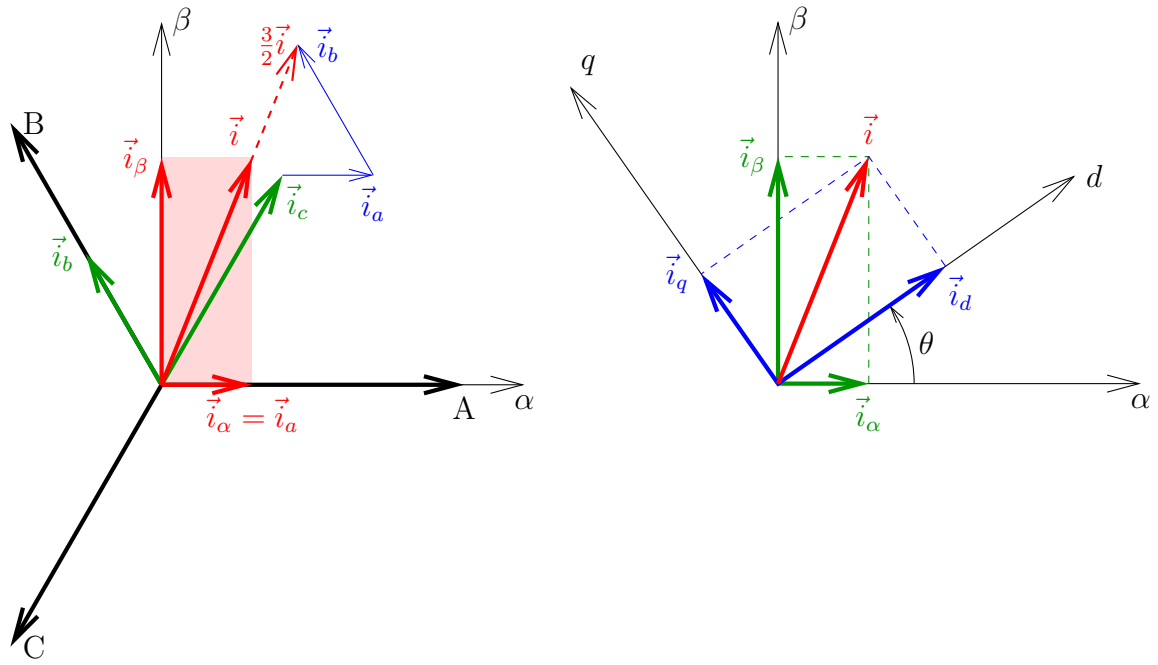
Při převodu se však běžně využívají dvě modifikace. Velikost vektorů na osách ABC odpovídá velikostem jednotlivých fázových proudů, obvykle normalizovaným do rozsahu -1 až 1. Při prostém vektorovém součtu by výsledný vektor měl velikost v rozsahu $-\frac{3}{2}$ až $\frac{3}{2}$. Během transformace je proto vektor zmenšen pomocí součinitele $\frac{2}{3}$, čímž vzniká tzv. amplitudově-invariantní Clarkeové transformace (*Clarke transformation*). V případě elektromotorů je navíc možné využít fakt, že prostý (ne vektorový) součet proudů všech tří fází je roven nule (předpokládá se ideální případ, kdy je odpor všech fází stejný). Výsledná transformace má potom podobu

$$\begin{bmatrix} \vec{i}_\alpha \\ \vec{i}_\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \vec{i}_a \\ \vec{i}_b \end{bmatrix} \tag{4.1}$$

Jedná se o běžně používané rovnice, dostupné například v [23; 24]. Transformace je znázorněna na obrázku 4.1a.

Další operací je převedení popisu ze statorového souřadného systému (α-β) do rotorového (d-q). Ten je definován pomocí vektoru magnetického indukčního toku $\vec{\psi}$ permanentních magnetů, na který se umístí osa d. K určení aktuální vzájemné polohy θ souřadných systémů α-β a d-q je proto zapotřebí znát aktuální úhel natočení rotoru. Pokud je známý, lze definovat transformaci

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} \tag{4.2}$$



(a) Clarkeové transformace [24] **(b)** Parkova transformace [24]

Obr. 4.1: Matematické transformace vektorového řízení

nazývanou Parkova transformace (*Park transformation*). Opět se jedná o běžně používané rovnice, které lze nalézt například v [25; 24]. Její geometrická interpretace je znázorněna na obrázku 4.1b. Nutno ještě podotknout, že úhel θ je tzv. elektrický úhel[3]. U vícepólových motorů se totiž rozlišují mechanický a elektrický úhel, které jsou vzájemně svázány vztahem

$$\theta_{el} = \frac{P}{2} \cdot \theta_{mech} \tag{4.3}$$

což souvisí s počtem period sinusovky napětí na jednu mechanickou otáčku (na každé fázi motoru). P je zde počet pólů motoru.

Výhoda Parkovy transformace je patrná z definice točivého momentu

$$\vec{M}_k = \frac{3}{2} \cdot \frac{P}{2} \cdot \vec{\psi} \times \vec{i} \tag{4.4a}$$

$$M_k = \frac{3}{2} \cdot \frac{P}{2} \cdot \psi \cdot i \cdot \sin\delta \tag{4.4b}$$

$$M_k = \frac{3}{2} \cdot \frac{P}{2} \cdot \psi \cdot i_q \tag{4.4c}$$

kde δ je úhel mezi vektorem proudu $\vec{i}$ a vektorem magnetického indukčního toku $\vec{\psi}$. Točivý moment tvoří pouze q složka, nazývaná momentotvorná, což výrazně usnadňuje jeho regulaci. Pro zajištění konstantního točivého momentu navíc stačí

[3]Pokud nebude specifikováno jinak, bude v této práci úhel θ vždy elektrický.

zajistit konstantní $\vec{i}_q$, k regulaci točivého momentu tak lze použít například obyčejný PI regulátor. [24]

Tokotvorná d složka proudu je u SPMSM (PMSM s magnety na povrchu rotoru, ostatními typy motorů se tato práce nezabývá) při nízkých otáčkách nežádoucí, neboť nemá vliv na výkon, ale zároveň zvyšuje spotřebu elektrické energie a zahřívá motor. Ve vysokých otáčkách se používá k odbuzování (oslabování) magnetického pole rotoru, aby bylo možné dosáhnout vyšších otáček [26].

Po provedení výpočtů regulace proudů $\vec{i}_d$ a $\vec{i}_q$ je nutné získaná napětí $\vec{v}_d$ a $\vec{v}_q$ opět převést do statorového souřadného systému α-β. K tomu slouží zpětná (inverzní) Parkova transformace

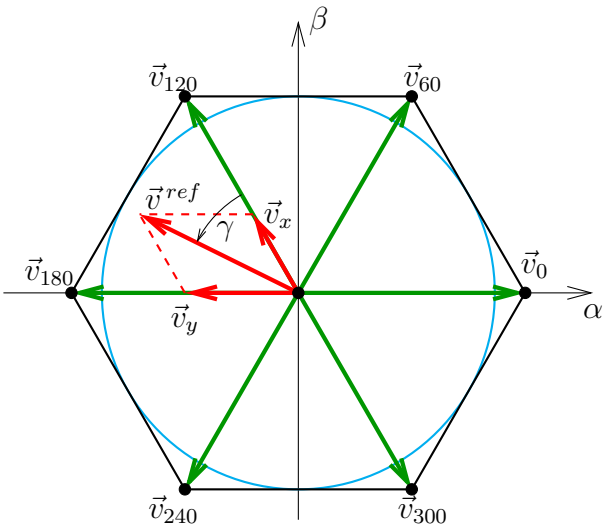
$$\begin{bmatrix} v_\alpha \\ v_\beta \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_d \\ v_q \end{bmatrix} \tag{4.5}$$

uvedená například v [27; 24].

4.2 Modulace napětí

Nyní, když je vektor napětí popsán složkami statorového souřadného systému, je nutné zajistit jeho promítnutí na jednotlivé fáze motoru. K tomu se využívá střídač, který připojuje jednotlivé fáze motoru na zdroj stejnosměrného napětí, nebo na nulu zdroje. Vzhledem k tomu, že pro chování motoru jsou důležitá sdružená napětí, nikoli fázová, existuje více možností, jak modulaci provést. Příklady napěťových modulací lze nalézt například v [28].

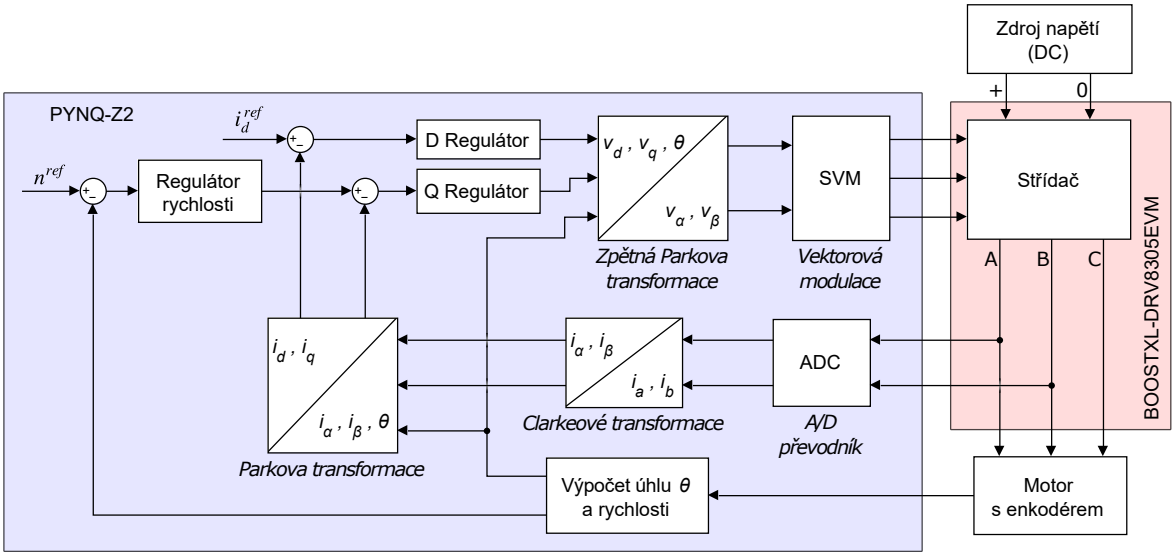
V této práci byla využita tzv. prostorově vektorová modulace (SVM, ***s**pace **v**ector **m**odulation*). Jedná se o symetrickou modulaci, která velmi efektivně využívá napětí zdroje (z hlediska maximální možné velikosti sdružených napětí). Využívá geometrické reprezentace sdružených napětí vznikajících při modulaci. Každá fáze motoru může být připojena buď na nulu, nebo napětí V_{DC} zdroje. To dává celkem 8 kombinací, které jsou vypsány v tabulce 4.1. Každá z těchto kombinací je reprezentována vektorem, jak je znázorněno na obrázku 4.2. V jedné periodě PWM se střídají 4 kombinace, přičemž 2 z nich patří nulovému vektoru $\vec{v}_{00}$ (nulová všechna sdružená napětí) a 2 některým dvěma vzájemně sousedícím napěťovým vektorům. Velikost vzniklých napěťových vektorů $\vec{v}_x$ a $\vec{v}_y$ je dána poměrem jejich časů v rámci dané periody PWM vůči času celé periody. Výsledný vektor $\vec{v}^{ref}$ je dán vektorovým součtem $\vec{v}_x$ a $\vec{v}_y$. Teoreticky lze vytvořit vektor napětí $\vec{v}^{ref}$ v rámci celého šestiúhelníku tvořeného základními vektory, prakticky se však využívá pouze oblast vyhraničená vepsanou kružnicí tohoto šestiúhelníku. Tomu odpovídá i výpočet, kde je amplituda vektoru napětí (původně



Obr. 4.2: Vektory napětí SVM [24]

Vektor	A	B	C
$\vec{v}_{00}$	0	0	0
	1	1	1
$\vec{v}_0$	1	0	0
$\vec{v}_{60}$	1	1	0
$\vec{v}_{120}$	0	1	0
$\vec{v}_{180}$	0	1	1
$\vec{v}_{240}$	0	0	1
$\vec{v}_{300}$	1	0	1

Tab. 4.1: Vektory napětí SVM [24]



Obr. 4.3: Schéma vektorového řízení [24] (upraveno)

v rozsahu -1 až 1) zmenšena koeficientem $\sqrt{3}/2$, aby se výsledný vektor do této kružnice vešel. [24]

4.3 Začlenění proudové regulace do zbytku algoritmu

Vektorové řízení v základu umožňuje efektivní řízení točivého momentu. V praxi je však často zapotřebí regulace rychlosti nebo polohy. V takovém případě je možné použít výstup rychlostní regulace jako referenci pro regulátor proudu $\vec{i}_q$. Další částí algoritmu také vždy musí být určení polohy. K tomu je možné využít senzor polohy, případně provádět její odhad (ze změřených proudů) za pomoci pozorovatele. Na obrázku 4.3 je znázorněno schéma řízení rychlosti při použití senzoru polohy.

5 Stanovení cílů

Nejdříve je nutné si stanovit cíle a požadavky na výsledné řízení, kterými jsou:

- Vytvoření základního vektorového řízení motoru Teknic M-2310P-LN-04K na platformě PYNQ-Z2
  - Regulace rychlosti
  - Uživatelsky nastavitelné parametry
  - Volitelná přímá regulace proudů (pro účely ladění regulátorů)
  - Nastavitelná frekvence PWM (volba z určitých hodnot)
- Umožnění určení polohy indexačního pulzu
- Umožnění zobrazování a ukládání měřených hodnot
- Vytvoření uživatelského rozhraní

Cílem praktické části této práce je vytvoření základního vektorového řízení a jeho uživatelského rozhraní. Uživatel by měl mít možnost ovládat motor (start/stop, volba žádané rychlosti) a nastavit parametry regulátorů. Frekvence PWM by měla být volitelná z určitého seznamu hodnot. Dále by mělo být umožněno zaznamenávání a průběžné zobrazování základních údajů o chování motoru (rychlost, proudy) a jejich následné uložení do souboru pro dodatečnou analýzu. Pro účely ladění proudových regulátorů by mělo být možné řízení přepnout do režimu proudové regulace s uživatelsky nastavitelnými žádanými hodnotami. Nakonec by měl být uživatel schopen s pomocí této aplikace určit polohu indexu enkodéru vůči rotorovému souřadnému systému d-q, jejíž znalost je pro řízení nezbytná (předpokládá se aktivní zapojení uživatele).

6 Použitý hardware a motor

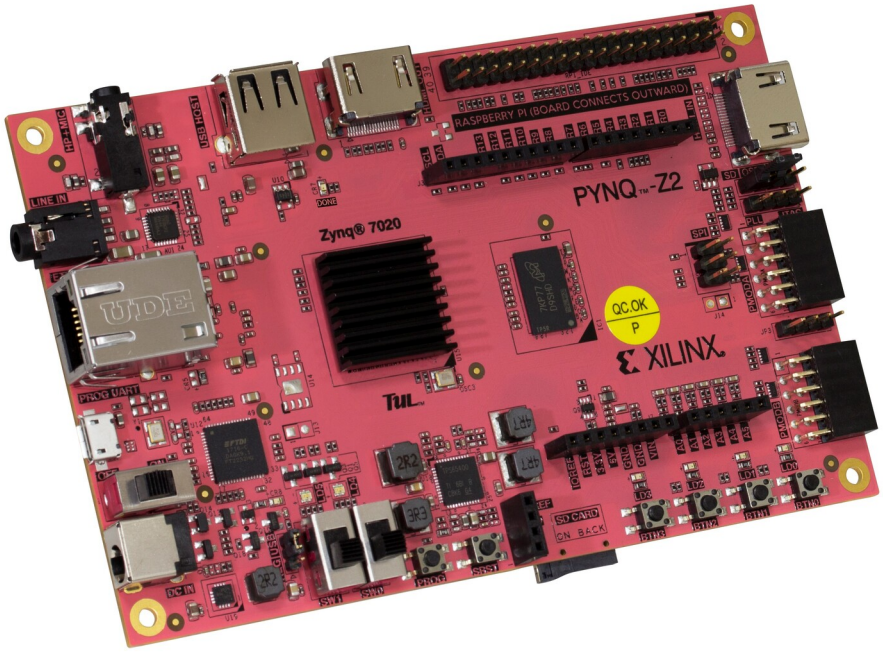
Tato práce je zaměřena na vývoj vektorového řízení elektromotoru Teknic M-2310P-LN-04K na platformě PYNQ-Z2. Jako výkonový modul byl použit BOOSTXL-DRV8305EVM. Tato kapitola se věnuje jejich základnímu popisu.

6.1 Vývojová deska PYNQ-Z2

Základem vývojové desky PYNQ-Z2 [29][4] je SOC (***s**ystem **o**n **c**hip*, systém na čipu) čip ZYNQ XC7Z020-1CLG400C [4] (zkráceně Zynq 7020), který v sobě kombinuje dvoujádrový ARM procesor a FPGA. V rámci FPGA je také integrován 12bitový A/D převodník. Přímo na desce je k dispozici 512 MB operační paměti DDR3, slot pro micro SD kartu, ethernetový konektor, 3,3 V vstupy a výstupy, 6 uživatelských LED diod (2 barevné), 4 uživatelská tlačítka a 2 uživatelské přepínače. K dispozici jsou ještě další konektory, které však nejsou pro tuto práci důležité. Vývojová deska je zobrazena na obrázku 6.1.

Deska je součástí projektu PYNQ [30; 31], jehož cílem je usnadnit vývoj aplikací na čipech kombinujících FPGA s mikroprocesorem. Pro tuto desku je k dispozici předkompilovaný obraz micro SD karty [32; 29], ze které je načítán operační systém s linuxovým jádrem. Po zapnutí je automaticky spuštěn server pro Jupyter Notebook a JupyterLab [33], který umožňuje přes webový prohlížeč vytvářet a spouštět

[4]Stránka [29] má v době psaní této práce prošlý certifikát protokolu *https*. Z tohoto důvodu je webovými prohlížeči blokována a je nutné přístup explicitně povolit (testováno v prohlížečích Google Chrome, Mozilla Firefox a Microsoft Edge).



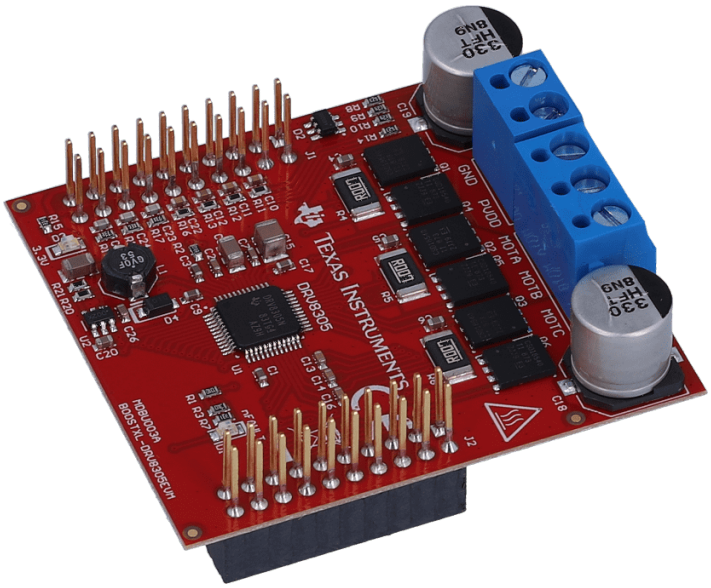
Obr. 6.1: Vývojová deska PYNQ-Z2 [29]

interaktivní dokumenty s Python kódem, stejně tak jako přistupovat k terminálu linuxu. V systému je předinstalován Python interpretr včetně PYNQ knihovny. Ta zajišťuje možnost nahrání konfigurace do FPGA přímo z Pythonu, stejně tak jako následnou komunikaci mezi Pythonem a podporovanými IP v FPGA. K terminálu linuxu je také možné přistupovat přes SSH, k uživatelskému adresáři v souborovém systému přes Samba protokol.

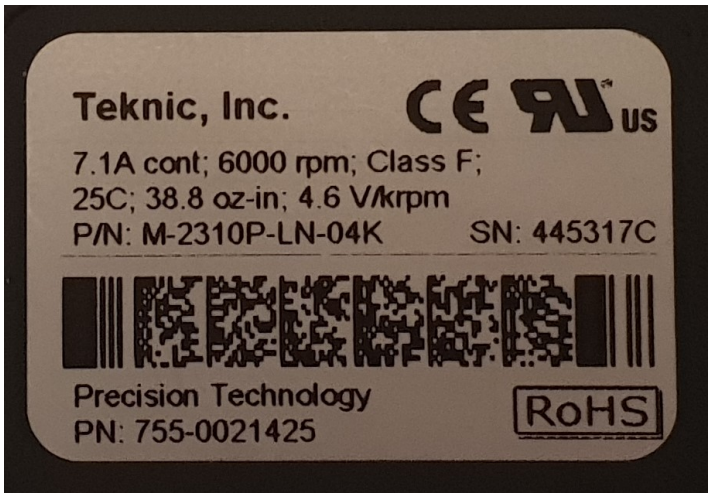
6.2 Výkonový modul BOOSTXL-DRV8305EVM

Výkonový modul BOOSTXL-DRV8305EVM [34], zobrazený na obrázku 6.2, je třífázový střídač, který zajišťuje napájení motoru z externího zdroje DC napětí. Modul je dimenzován na napětí 4,4 až 45 V a proud až 15 A RMS (efektivní hodnota). Základem modulu je ovladač výkonových tranzistorů DRV8305, který řídí operaci celého modulu. Ovládání výkonových tranzistorů je prováděno externími 3,3 V PWM signály (LVTTL), přivedenými na vstupní piny modulu. Modul také umožňuje měření napětí a proudů na jednotlivých fázích, stejně tak jako napětí zdroje.

K měření zmíněných napětí jsou na modulu dostupné napěťové děliče, jejichž výstupy jsou vyvedeny na výstupní piny modulu. Měření proudů je zajištěno pomocí měřicích odporů, umístěných mezi dolními výkonovými tranzistory a nulou zdroje. Napětí na nich vznikající jsou v rámci DRV8305 zesílena operačním zesilovačem a následně opět vyvedena na výstupní piny [35; 36]. Tento způsob zapojení umožňuje měření proudu, pokud je příslušná fáze sepnuta na nulu.



Obr. 6.2: Výkonový modul BOOSTXL-DRV8305EVM [34]



Obr. 6.3: Štítek motoru

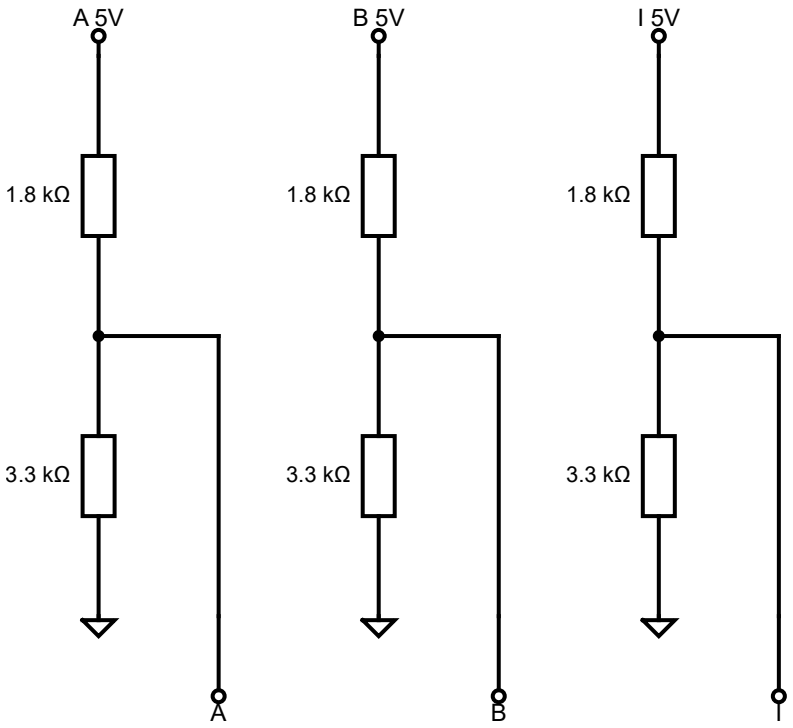
Jmenovité otáčky	n_n	6 000	RPM
Jmenovitý proud	I_n	7,1	A
Odpor vinutí (fáze vůči fázi)	R	0,72	Ω
Indukčnost vinutí (fáze vůči fázi)	L	0,40	mH
Elektrická časová konstanta	τ_e	0,56	ms
Zpětné elektromotorické napětí	K_e	4,64	$\frac{\mathrm{V_{peak}}}{\mathrm{kRPM}}$

Tab. 6.1: Parametry motoru

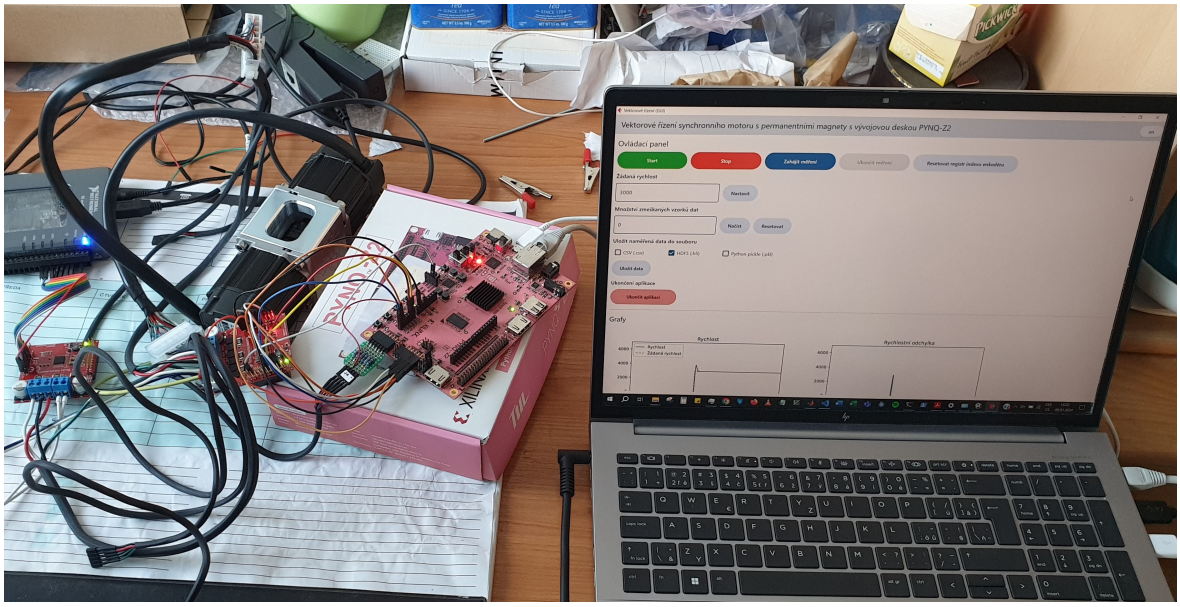
6.3 Motor Teknic M-2310P-LN-04K

Motor Teknic M-2310P-LN-04K [37; 38] je 8pólový (4 pólpáry) bezkartáčový synchronní motor s permanentními magnety. Základní parametry motoru jsou v tabulce 6.1, fotografie štítku na obrázku 6.3. Motor obsahuje vestavěný kvadratický inkrementální enkodér se 4 000 polohami na mechanickou otáčku (1 000 poloh na elektrickou otáčku). Enkodér je napájen 5 V, digitální vstupy na desce PYNQ-Z2 však vyžadují 3,3 V. Z tohoto důvodu je nutné začlenit převodník napěťových úrovní, viz schéma na obrázku 6.4. Zdroj 5 V je na desce přímo k dispozici.

Fotografie pracoviště je na obrázku 6.5.



Obr. 6.4: Schéma napěťového převodníku



Obr. 6.5: Fotografie pracoviště

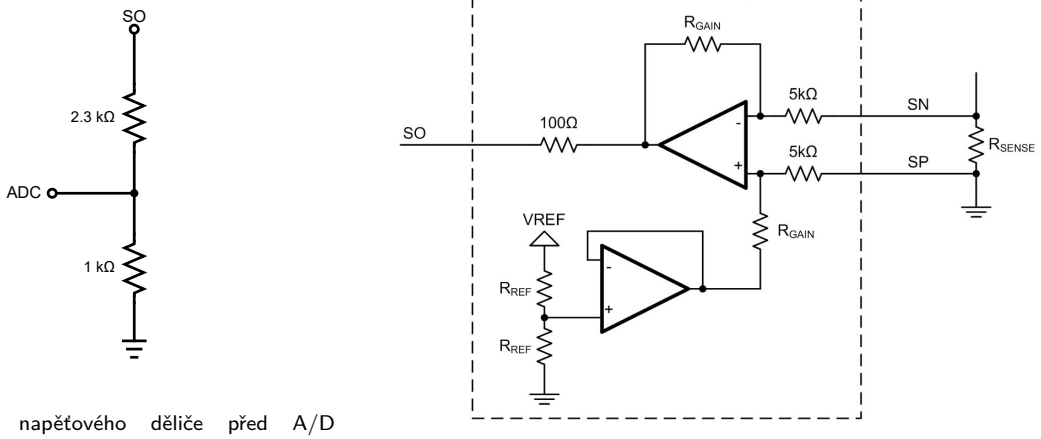
7 Realizace řídicího algoritmu

V rámci této kapitoly bude popsána realizace řídicího algoritmu. Ten byl celý realizován v rámci FPGA. Návrh byl proveden v prostředí Simulink, exportován do Verilogu a následně syntetizován a implementovám v prostředí Vivado. Oba projekty jsou k dispozici v příloze této práce. Model ze Simulinku byl navíc exportován tak, aby bylo možné jej prohlížet bez nutnosti instalace Simulinku, viz kapitola 10.

Návrh byl proveden na úrovni RTL a celý model byl exportován do Verilogu jako jeden modul (generovaný kód je samozřejmě dále členěn na jednotlivé moduly podle struktury modelu, z hlediska integrace do projektu ve Vivado se však jedná o jeden celek). Jako základní frekvence bylo zvoleno 40 MHz, nižší frekvence byly použity pro komunikaci s mikroprocesorem (10 MHz) a výpočet převrácené hodnoty při výpočtu rychlosti (5 MHz). V nastavení generování HDL kódu bylo zvoleno "více zdrojů hodinových signálů". K jejich odvození ze základní frekvence poskytované procesorem bylo využito specializované IP *Clocking Wizard* [39] dostupné v rámci základní instalace Vivado.

7.1 Měření fázových proudů

Jak již bylo zmíněno v kapitole 6.2, výkonový modul přímo umožňuje měření proudů v jednotlivých fázích, viz schéma na obrázku 7.1b. Měřicí odpory R_{SENSE} mají velikost 7 mΩ a jsou umístěny mezi nulou zdroje a tranzistorem, který danou fázi na nulu připojuje. V závislosti na protékajícím proudu (včetně orientace) vzniká na odporu napětí podle Ohmova zákona, které je následně upraveno v rámci čipu DRV8305. Ve výchozím nastavení čipu je toto napětí zesíleno 10krát a následně je k němu přičtena polovina referenčního napětí *VREF*, které je na modulu pevně nastaveno na hodnotu 3,3 V. Tím je zajištěna možnost měřit proudy v obou směrech pomocí A/D převodníků pracujících s napětím 0 až 3,3 V. Analogové zesílení napětí je zde z důvodu



(a) Schéma napěťového děliče před A/D převodníkem (zjednodušené schéma)

(b) Schéma obvodu pro měření fázových proudů [35]

Obr. 7.1: Měření fázových proudů

zvýšení přesnosti měření, zajišťuje totiž lepší využití rozsahu napětí měřitelných A/D převodníkem. Zvolená velikost měřicího odporu R_{SENSE} zde v kombinaci s výchozím zesílením napětí mapuje proudy, na které je modul dimenzovaný, na napětí v rozsahu 0 až 3,3 V vztahem

$$I = \frac{V_O - 1,65}{R_{SENSE} \cdot 10} = \frac{V_O - 1,65}{0,007 \cdot 10} = \frac{V_O - 1,65}{0,07} \tag{7.1}$$

kde V_O je napětí, které se objeví na výstupním pinu modulu, reprezentující daný proud I. Za kladný je zde považován proud, který přitéká danou fází do motoru (rozdíl napětí $SP - SN$ je kladný), viz dokumentace [36].

A/D převodník dostupný v rámci čipu Zynq 7020 umožňuje měření v rozsahu 0 až 1 V, na desce PYNQ-Z2 je proto (z důvodu kompatibility) připojen přes napěťový dělič, viz obrázek 7.1a (plné schéma je k dispozici v dokumentaci [40] dostupné na stránce [29]). Ten je však ovlivněn odporem 100 Ω, který se nachází za operačním zesilovačem. Proto je nutné provést korekci měřeného napětí koeficientem G_{corr} vycházejícím ze soustavy rovnic

$$V_{OZ} = (100 + 2320 + 1000) \cdot I \tag{7.2a}$$

$$V_{ADC} = 1000 \cdot I \tag{7.2b}$$

$$\frac{V_{ADC} \cdot G_{corr}}{V_{OZ}} = \frac{1}{3,3} \tag{7.2c}$$

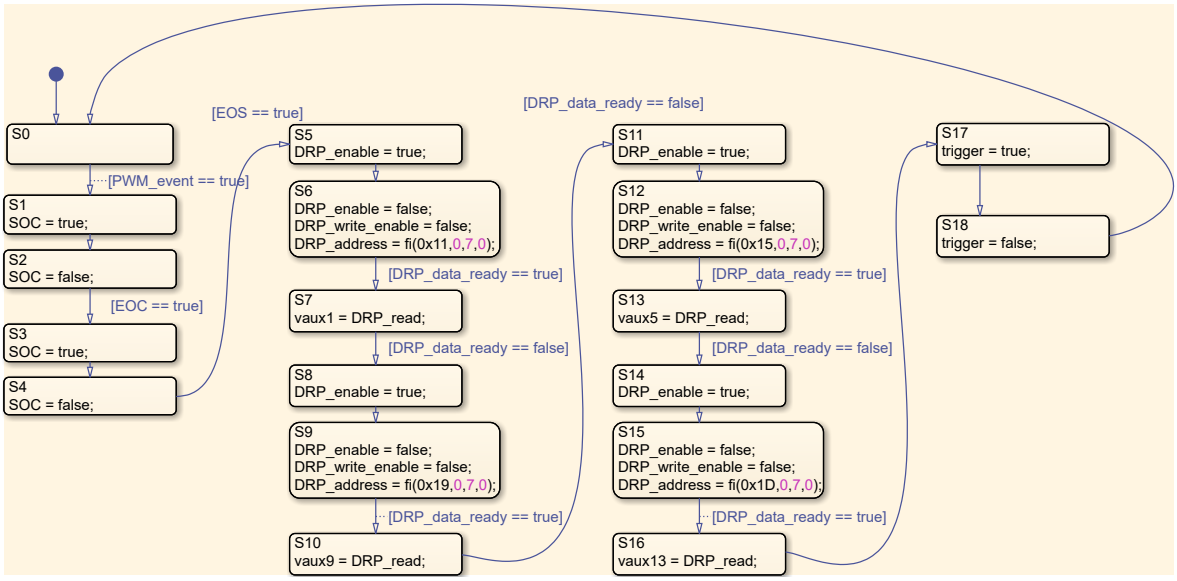
kde V_{OZ} je napětí na výstupu operačního zesilovače (před 100Ω odporem), V_{ADC} je napětí na vstupu do A/D převodníku a I je protékající proud. Velikost korekčního koeficientu vychází

$$G_{corr} = \frac{V_{OZ}}{3,3 \cdot V_{ADC}} = \frac{100 + 2320 + 1000}{3,3 \cdot 1000} = 1,036 \tag{7.3}$$

Výpočet zanedbává vliv samotného A/D převodníku a s ním spojených odporů.

Čip Zynq 7020 obsahuje 1 konfigurovatelný dvoukanálový A/D převodník *XADC* [41]. Jeho konfigurace probíhá ve Vivado pomocí IP *XADC Wizard* [42]. Umožňuje měření 2 napětí současně, přičemž je možné ho připojit na různé vstupní piny a průběžně je střídat. Na vývojové desce PYNQ-Z2 jsou k dispozici celkem 4 piny (2 páry) podporující současné měření. Tyto piny jsou v této práci využity k měření všech tří fázových proudů a napětí zdroje (současné měření proudů ve fázích A a B je žádoucí).

Zahájení konverzí a čtení změřených hodnot je prováděno v rámci FPGA. Za tímto účelem byl vytvořen jednoduchý stavový diagram zobrazený na obrázku 7.2. Měření je prováděno vždy úplně na začátku periody PWM, který je indikován pulzem



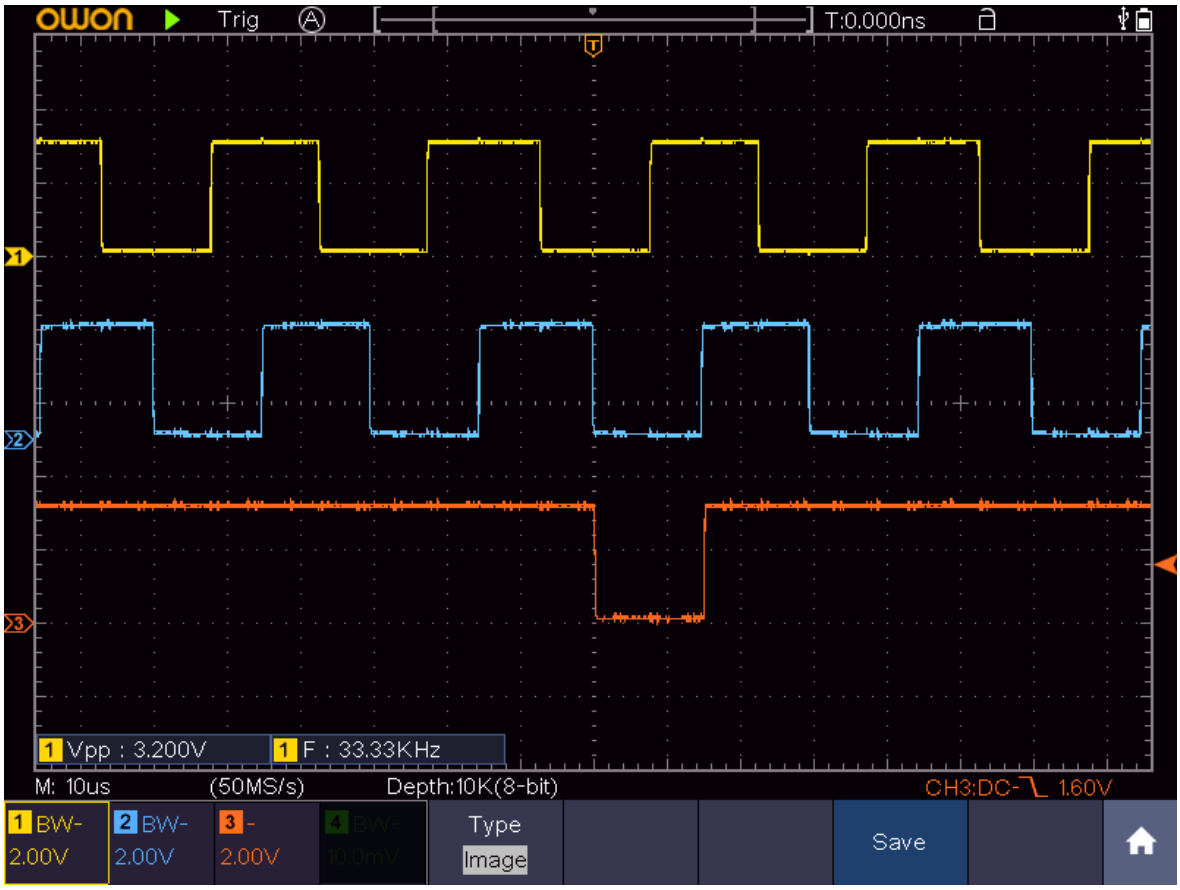
Obr. 7.2: Stavový diagram pro ovládání A/D převodníku vytvořený pomocí toolboxu Stateflow. V hranatých závorkách jsou uvedeny podmínky přechodu. V každém časovém kroku může dojít maximálně k jednomu přechodu mezi stavy, nehledě na přítomnost nebo platnost podmínky dalšího přechodu. Počáteční hodnoty všech výstupů jsou nulové. Adresy ADC registrů *DRP_address* jsou 7bitová (3. argument funkce *fi*) nezáporná (2. argument) celá (4. argument – počet desetinných míst) čísla v diagramu zapsaná hexadecimálně (1. argument).

PWM_event. V tuto chvíli je inicializován začátek konverze (*SOC*) proudů ve fázích A a B. Po dokončení konverzí (*EOC*) je zahájeno měření proudu ve fázi C a napětí zdroje. Po dokončení celé sekvence měření (*EOS*) jsou přečteny výsledky z příslušných registrů pomocí paralelní DRP (***D****ynamic* ***R****econfiguration* ***P****ort*) komunikace. Nakonec je vytvořen řídicí pulz *trigger*, který putuje mezi RTL registry společně s platnými daty. Tento pulz trvá jednu periodu hodinového signálu (25 ns) a zajišťuje správné chování některých bloků (například integrátory regulátorů nesmí integrovat každou periodu hodinového signálu, ale pouze jednou za periodu PWM).

Změřené hodnoty je následně nutné převést do požadovaného rozsahu -1 až 1. Vzhledem k tomu, že A/D převodník je 12bitový, nacházejí se změřené hodnoty (po extrahování z 16bitových registrů) v rozsahu 0 až 4 095. Převod je proveden pomocí uživatelem volitelných parametrů vztahem

$$I_{out} = ((I_{in} + C_{ADC\_zero\_offset}) \cdot G_{corr}) - C_{I\_zero\_offset} \tag{7.4}$$

kde I_{in} je hodnota před převodem a I_{out} hodnota po převodu. G_{corr} slouží ke korekci měření (odstranění nežádoucího vlivu napěťového děliče) podle rovnice (7.3). $C_{I\_zero\_offset}$ slouží k odstranění napěťového posunutí 1,65 V vytvořeného na výkonovém modulu, výchozí hodnota je 2 048 (polovina rozsahu). $C_{ADC\_zero\_offset}$ umožňuje případnou kompenzaci napěťových rozdílů mezi nulou A/D převodníku a nulou výkonového modulu. Tento parametr může být kladný i záporný, výchozí hodnota



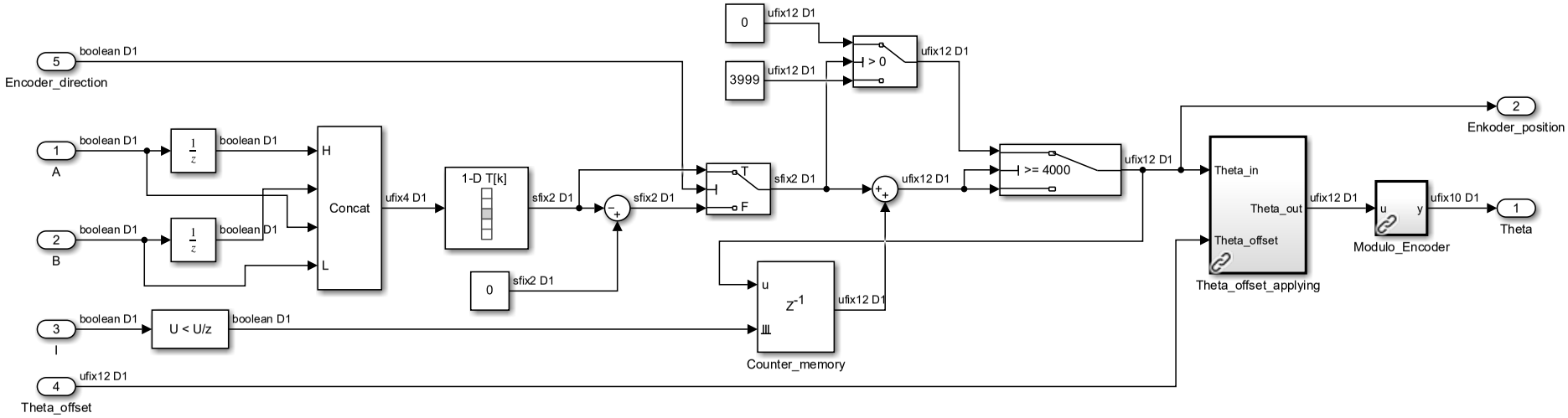
Obr. 7.3: Signály použitého kvadratického inkrementálního enkodéru měřené na osciloskopu při rychlosti motoru 2000 $\frac{ot}{min}$. Žlutě je kanál *A*, modře kanál *B* a oranžově indexační kanál *I*.

je nula. Všechny tyto parametry jsou uživatelsky volitelné pro každý fázový proud samostatně, což umožňuje případné kompenzace nepřesností parametrů hardwarových prvků.

7.2 Měření polohy

Určování polohy a rychlosti bylo realizováno pomocí inkrementálního kvadratického enkodéru integrovaného v motoru. Vzhledem k tomu, že výstupem enkodéru jsou tři digitální signály A, B a I a maximální frekvence podporovaná enkodérem je 667 kHz [37] (frekvence rozlišitelných změn polohy, frekvence signálů na jednotlivých kanálech A a B je čtvrtinová), je možné vytvořit vyhodnocovací logiku kompletně v rámci FPGA (40 MHz). Jedinou podmínkou je již zmíněný převod napětí z 5 V na 3,3 V, viz kapitola 6.3.

Na obrázku 7.3 jsou zobrazeny signály použitého enkodéru při otáčkách 2 000 $\frac{ot}{min}$. Kanály A a B mají každý 1 000 pulzů na otáčku a jsou vzájemně fázově posunuty o 90°. Fázové posunutí umožňuje určení směru otáčení. Při sledování náběžných i spádových hran obou kanálů je rozlišení enkodéru 4 000 poloh na mechanickou otáčku. Vyhodnocování přírůstku polohy je provedeno formou pravdivostní tabulky. Mírnou



Obr. 7.4: Vyhodnocování signálů enkodéru

komplikací zde je, že počet poloh na otáčku není mocninou 2. Z tohoto důvodu musely být přidány podmínky zajišťující přetečení čítače z 3 999 na 0 a obráceně, viz schéma na obrázku 7.4. Čítač je také resetován na nulu pomocí indexačního pulzu na kanále I, který se objeví jednou za otáčku. Aktuální poloha enkodéru je vyvedena do uživatelského rozhraní, aby bylo možné určit polohu θ_{offset} rotorového souřadného systému d-q vůči indexu. Úhel natočení rotoru θ_{mech} je pak získán odečtením úhlu θ_{offset} od polohy enkodéru, opět je nutné zajistit správné přetečení mezi hodnotami 0 a 3 999. Úhel θ_{el} mezi statorovým souřadným systémem α-β a rotorovým souřadným systémem d-q je získán jako zbytek po dělení (modulo) úhlu θ_{mech} číslem 1 000. Je to z toho důvodu, že motor má 4 pólpáry a na jednu elektrickou otáčku tak připadá 1 000 poloh enkodéru. Dělení a s ním spojený výpočet modulo jsou však na FPGA velmi problematické (dlouhá doba výpočtu a vysoké využití prvků, výjimkou je dělení mocninou 2, které odpovídá bitovému posunu), k určení zbytku po dělení tak byla využita série odečítání a porovnávání (schéma dostupné v příloze).

Celá logika zobrazená na obrázku 7.4 je vyhodnocena v jednom časovém kroku. Před tímto subsystémem i za ním jsou vloženy registry. Obecně jsou v celém návrhu přidávány registry mezi digitální vstupy/výstupy z FPGA a příslušnou logiku v rámci FPGA (v tomto případě vstupy A, B a I).

Čítač polohy enkodéru má po spuštění algoritmu hodnotu 0, což neodpovídá realitě. Proto je nutné před zahájením řízení manuálně pootočit motorem, dokud se neobjeví indexační pulz, který už polohu přesně definuje. Z tohoto důvodu byl také vytvořen binární registr s počáteční hodnotou 0, který je nastaven na 1 při prvním výskytu indexačního pulzu a tuto hodnotu již neustále drží. Hodnota tohoto registru je zobrazena pomocí LED diody LD3 přímo na desce PYNQ-Z2 – dioda se po prvním výskytu pulzu rozsvítí. Registr je navíc možné resetovat na nulu pomocí uživatelského rozhraní. To se může hodit, pokud bylo z nějakého důvodu nutné odpojit enkodér motoru od desky a znovu připojit.

Určení polohy je v celém řízení důležité ze dvou důvodů – využívá se pro výpočet rychlosti a pro Parkovu transformaci (včetně té zpětné). Parkova transformace však vyžaduje sinus a kosinus úhlu θ, což jsou výpočetně také celkem náročné funkce. Jejich hodnoty jsou proto pro všechny možné vstupy (0 až 999) předem vypočítány v tabulkách (každá funkce v jedné tabulce). Tyto tabulky jsou v rámci FPGA realizovány pomocí BRAM. Toho lze docílit použitím tabulky (LUT) v Simulinku, za kterou ihned následuje neresetovatelné jednotkové zpoždění (registr), viz [43].

7.3 Výpočet rychlosti

Od řízení se nevyžaduje polohová regulace a s ní související určování rychlosti při velmi malých otáčkách. Naopak je vhodné mít rychlost nějakým způsobem filtrovanou. Určení rychlosti je proto realizováno formou průměrné rychlosti za jednu mechanickou otáčku a je aktualizováno každých 45° (mechanických). Průměrná rychlost je počítána na základě rozdílu časů, ve kterých se motor v dané poloze vyskytl. Mimo jiné za tímto účelem byl vytvořen 64bitový čítač cyklů hlavního hodinového signálu (40 MHz). Tímto způsobem je možné počítat rychlost s relativně vysokým rozlišením v porovnání s výpočtem založeným na sledování počtu změn polohy za stanovený čas. Zároveň je tento způsob také výpočetně komplikovanější, protože vyžaduje dělení nekonstantní hodnotou (dělení předem známou konstantní hodnotou je možné převést na násobení předem známou převrácenou hodnotou, které není problematické). Ani využití tabulky, jako v případě sinu a kosinu, zde není možné, protože by byla příliš veliká. Dělení je nicméně rozděleno na výpočet převrácené hodnoty dělitele a následné násobení. Převrácenou hodnotu s danou přesností není možné na použitém hardwaru realizovat za 25 ns (40 MHz). Proto je výpočet převrácené hodnoty převeden na frekvenci 5 MHz (200 ns).

Výpočet rychlosti provedený tímto způsobem také neobsahuje informaci o směru, ten je nutné určit samostatně. Za tímto účelem byl vytvořen algoritmus, který vyhodnotí směr pokaždé, když se θ změní o 4 polohy enkodéru (případně násobek 4, pokud by z nějakého důvodu došlo k dočasnému výpadku signálů enkodéru). Vyhodnocení totiž proběhne pokaždé, když se zbytek po dělení úhlu θ číslem 4 rovná nule a zároveň se θ liší od úhlu θ, ve kterém bylo vyhodnocení provedeno naposledy. V tomto případě je zmíněný zbytek po dělení roven posledním dvěma bitům (na straně LSB – 2 nejméně významné bity) úhlu θ, což umožňuje jeho jednoduché určení.

Při vyhodnocování směru je také vždy uložen aktuální čas, takže je možné velmi rychle zjistit, že se motor zastavil. Při vyhodnocování rychlosti je nutné nějakou dobu počkat, než je možné dojít k tomuto závěru. Čím menší je minimální měřitelná rychlost a čím méně často dochází k vyhodnocování, tím delší tato doba je. Vyhodnocování rychlosti však probíhá pouze každých 45° (500 poloh enkodéru). Vyhodnocování směru, které probíhá každé 4 polohy enkodéru, je proto mnohem lepším kandidátem. Pokud nedojde k pootočení motoru o 4 polohy (v souladu s jejich výše popsaným vyhodnocením) za 65 535 hodinových cyklů (přibližně 1,6 ms), je rychlost považována za nulovou. To odpovídá minimální měřitelné rychlosti přibližně 37 $\frac{ot}{min}$.

7.4 Regulace

K regulaci proudů i_d a i_q byly použity PI regulátory v paralelní formě. Stejný typ regulátoru byl použit i pro regulaci rychlosti. Základní funkci těchto regulátorů v diskrétním stavu lze popsat rovnicí

$$u_i = K_P \cdot e_i + \sum_{k=0}^{t_i} (T \cdot K_I \cdot e_i) \tag{7.5}$$

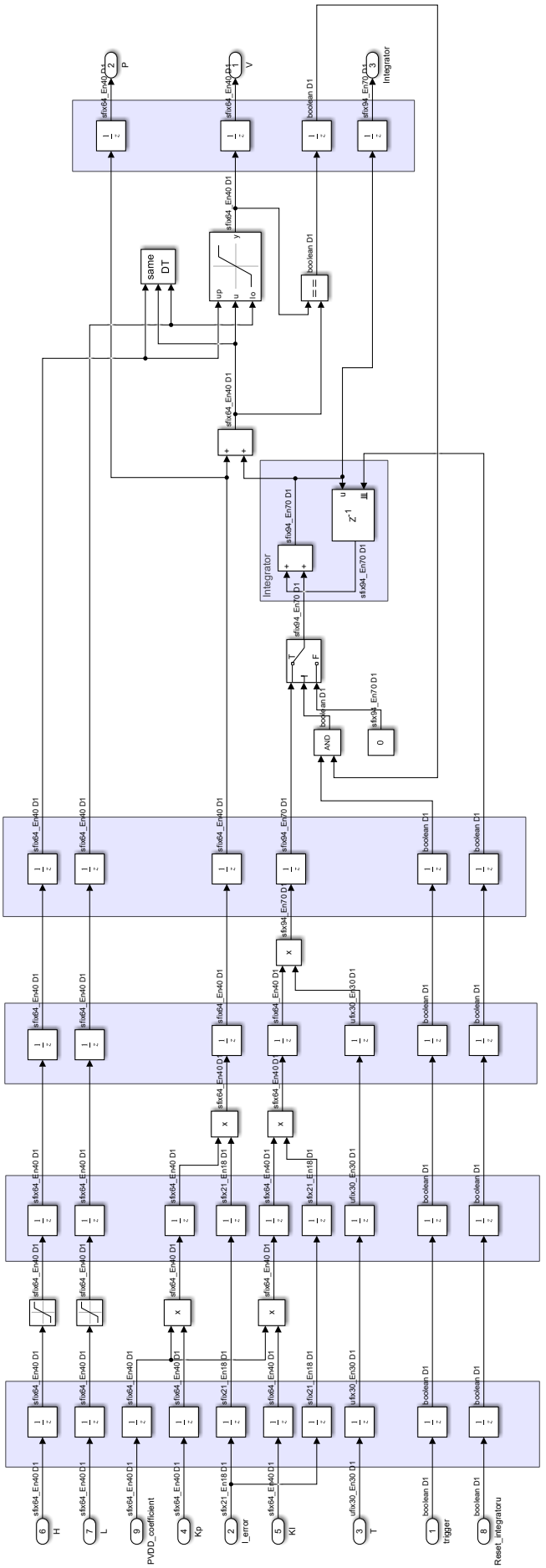
kde u_i je aktuální akční zásah, e_i aktuální regulační odchylka, K_P konstanta proporcionální složky, K_I konstanta integrační složky a T perioda PWM. Suma zde představuje diskrétní náhradu integrátoru a její vyhodnocení probíhá jednou za periodu PWM (pokaždé, když jsou k dispozici data z nového měření A/D převodníku).

Regulátory jsou opatřeny omezením akčního zásahu a anti-windupem. Anti-windup je zde řešen metodou zastavení integrace – pokud dojde k omezení akčního zásahu, přestane se integrační složka měnit a ponechá si svou hodnotu až do doby, kdy proporcionální složka klesne a omezení přestane být aktivní. Integrační složku je navíc možné resetovat na nulu, aby nedocházelo k jejímu vyhodnocování, když daná regulace není zapnuta. V případě proudových regulátorů je to při vypnutém PWM. V takovém případě totiž výkonový modul BOOSTXL-DRV8305EVM neumožňuje čtení proudů a na příslušných pinech je nulové napětí. To způsobuje, že měřené hodnoty všech fázových proudů jsou na svých číselných maximech (z hlediska absolutní hodnoty), tj. přes -20 A.

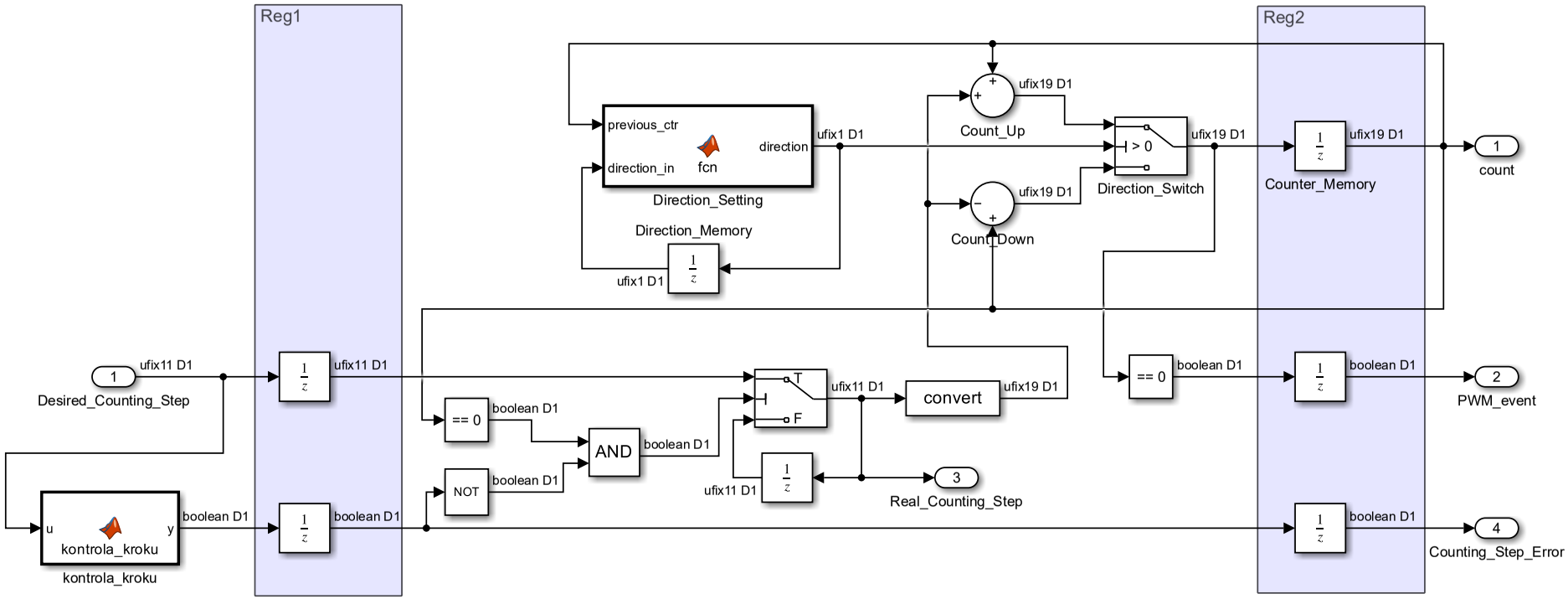
Realizace proudových regulátorů je zobrazena na obrázku 7.5. Ty jsou navíc opatřeny volitelným přepočtem parametrů K_P a K_I, který zohledňuje měřené napětí zdroje a umožňuje tak částečně eliminovat změny regulačních vlastností způsobené změnou zdrojového napětí. Veškeré parametry jsou uživatelsky nastavitelné.

7.5 Generování PWM

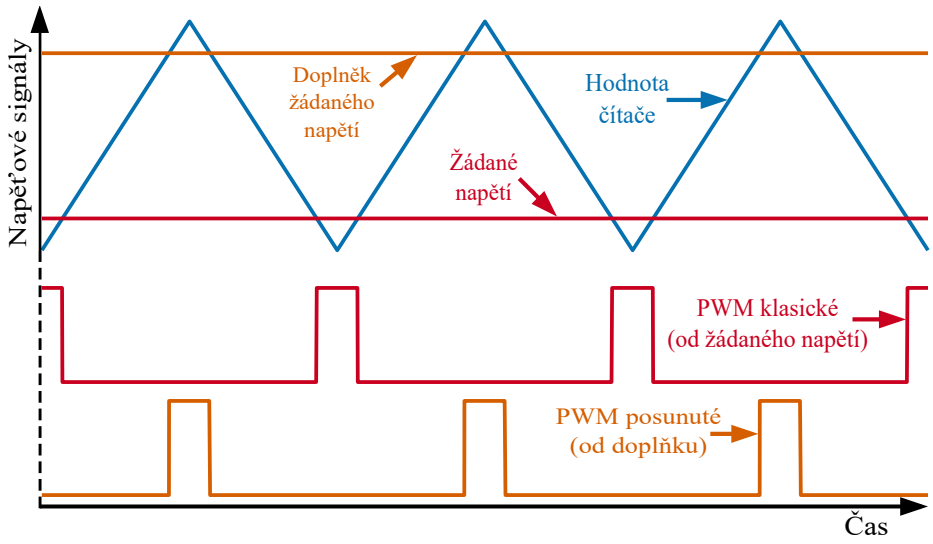
Základem generování pulzně šířkové modulace je čítač hodinového signálu. V případě symetrické PWM čítá od 0 do určité hodnoty, kde změní směr a čítá zpět do nuly. Tento proces se neustále opakuje a jedno opakování je považováno za periodu PWM. Realizace čítače je znázorněna na obrázku 7.6. Maximální hodnota čítače $count_{max}$ byla nastavena na hodnotu 360 000, kvůli její dobré dělitelnosti. Frekvence PWM je nastavována velikostí kroku čítače *counting_step*. Případná změna frekvence se uloží, když je čítač roven nule. Krok čítače také musí být zvolen tak, aby s ním bylo možné dělit maximální hodnotu čítače beze zbytku. Frekvenci PWM je pak možné určit



Obr. 7.5: Proudový regulátor



Obr. 7.6: Obousměrný čítač pro generování PWM



Obr. 7.7: Princip generování PWM. Nahoře je zobrazen časový průběh hodnoty čítače a žádané hodnoty napětí a jejího doplňku. Hodnoty napětí jsou přepočítané do rozsahu hodnot čítače. Pod tím jsou zobrazeny průběhy příslušných PWM signálů vzniklých porovnáváním žádané hodnoty (respektive jejího doplňku) a hodnoty čítače.

vztahem

$$f_{PWM} = \frac{counting\_step}{2 \cdot count_{max} \cdot T_{clk}} = \frac{counting\_step}{2 \cdot 360\,000 \cdot 25 \cdot 10^{-9}} = \frac{counting\_step}{0,018} \tag{7.6}$$

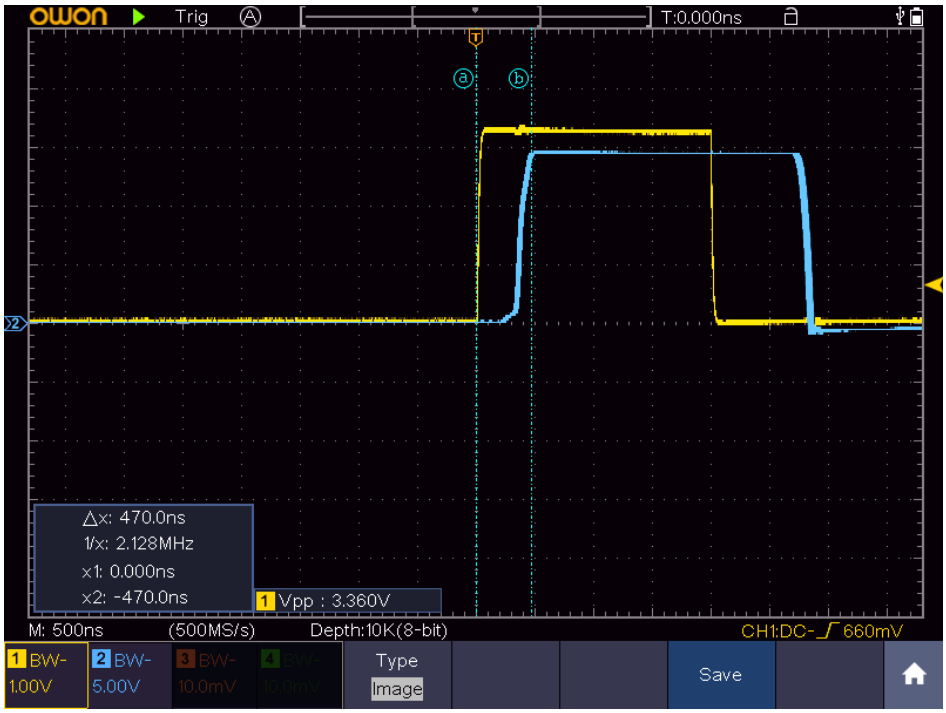
kde T_{clk} je perioda hodinového signálu.

Výsledný průběh napětí na dané fázi vzniká tak, že se neustále porovnává hodnota čítače a žádaného napětí dané fáze. Pro tento účel musí být požadované napětí přepočítáno do rozsahu 0 až 360 001. Pokud je požadované napětí ostře větší než hodnota čítače, je daná fáze připojena na napětí zdroje, v opačném případě je sepnuta na nulu. Postup je také možné modifikovat tak, že se čítač porovnává s doplňkem požadovaného napětí

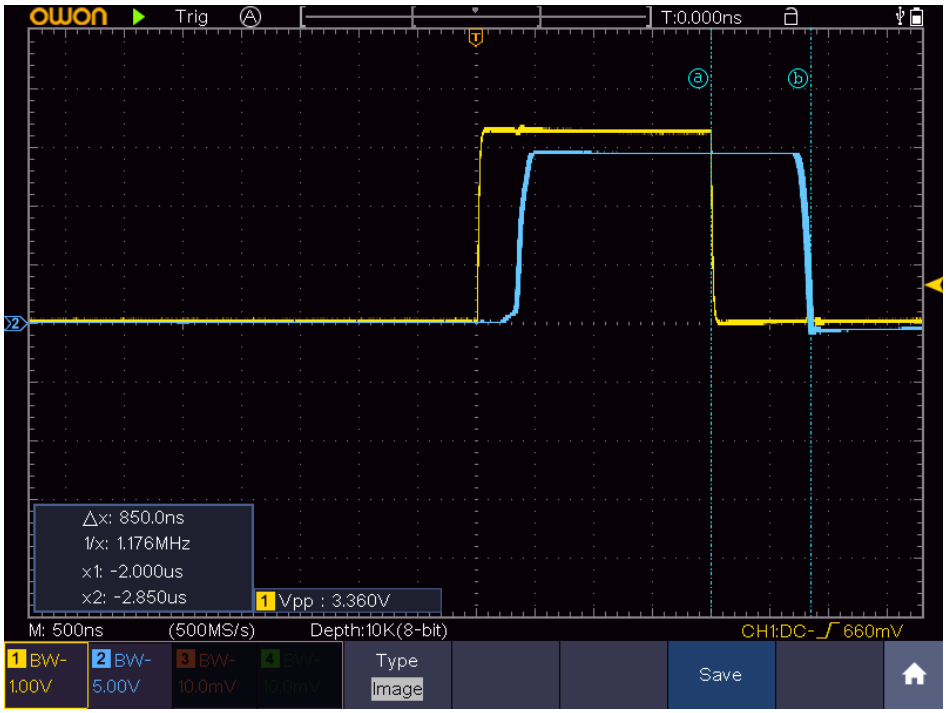
$$v_{complement} = 360\,001 - v \tag{7.7}$$

V tomto případě musí být hodnota doplňku naopak ostře menší, než je hodnota čítače, aby byla fáze sepnuta na napětí zdroje. Tato úprava posune tvar PWM signálu o polovinu periody, na výsledné chování nemá vliv, viz obrázek 7.7. V případě prostorově vektorové modulace však toto posunutí zapříčiní, že se při nulové hodnotě čítače nachází všechny fáze sepnuté na nulu. To je vyžadováno pro měření proudů, neboť měřicí odpory jsou umístěny mezi dolními tranzistory a nulou zdroje. Druhou možností by bylo provádět měření v polovině periody PWM.

Zvětšení rozsahu hodnot napětí o 1 v porovnání s rozsahem hodnot čítače je zde z toho důvodu, aby bylo možné na dané fázi požadovat neustále sepnutý stále stejný výkonový tranzistor (horní nebo dolní). Při stejných rozsazích by v závislosti na ostrosti porovnání bylo možné tohoto docílit pouze pro jeden z nich. V rámci realizovaného řízení tato



Obr. 7.8: Prodleva náběžné hrany výkonového tranzistoru. Žlutě je zobrazen řídicí PWM signál a modře napětí za horním výkonovým tranzistorem.



Obr. 7.9: Prodleva spádové hrany výkonového tranzistoru. Žlutě je zobrazen řídicí PWM signál a modře napětí za horním výkonovým tranzistorem.

funkce sice nakonec nebyla využita (stačila by možnost držet neustále sepnutý dolní tranzistor), zahrnutí této drobnosti však zlepšuje opětovnou využitelnost realizovaného generátoru PWM.

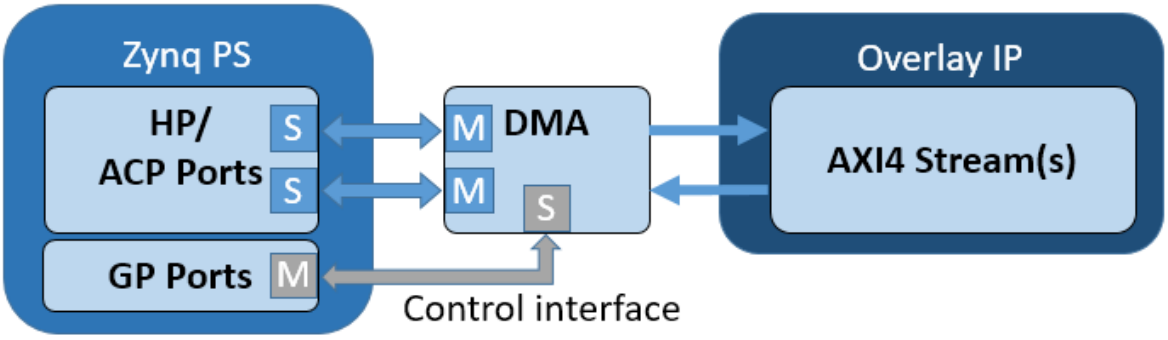
Při generování PWM je také nutné vytvořit tzv. mrtvou dobu. Výkonové tranzistory mají určité zpoždění, než sepnou či rozepnou. Tyto prodlevy byly u použitého výkonového modulu experimentálně proměřeny, výsledky jsou zobrazeny na obrázcích 7.8 a 7.9. Z měření je patrné, že prodleva u spádové hrany je delší než prodleva u náběžné hrany. Aby nedocházelo ke zkratu, je zapotřebí přidat prodlevu (mrtvou dobu) mezi rozepnutí jednoho tranzistoru dané fáze a sepnutí druhého. Tato prodleva byla nastavena s rezervou na 1 µs.

Nakonec byla přidána možnost generovat uživatelem zadanou střídu PWM na fázi A při současném ponechání zbylých fází na nule. Tato možnost slouží k určení polohy rotorového souřadného systému vůči indexu enkodéru.

7.6 Komunikace mezi FPGA a mikroprocesorem

Interakci FPGA s procesorovým systémem rodiny Zynq 7000 zajišťuje ve Vivado IP *Zynq 7000 Processing System* [44]. Kromě zdroje hodinového signálu zajišťuje také komunikaci pomocí AXI protokolu a vyvolání přerušení v mikroprocesoru. Pro práci s AXI komunikací existují ve Vivado různá IP, některá z nich byla již ukázána na obrázku 3.2 v kapitole 3.1. Jedním z nich je *AXI GPIO* [45], které umožňuje přečíst nebo zapsat hodnotu určitého registru o velikosti 1 až 32 bitů. Z hlediska komunikace se chová jako slave, je tedy možné ho ovládat programem běžícím na mikroprocesoru. To je využito například k nastavování žádané rychlosti, spouštění řízení nebo čtení aktuální polohy enkodéru.

Tento postup však není vhodný pro přenášení většího množství dat, obzvláště pokud se jedná o vysokorychlostní přenos. Za tímto účelem bylo využito IP *AXI Direct Memory Access* (zkráceně DMA) [46]. Prostřednictvím IP Processing System je totiž možné přistupovat přímo do paměti mikroprocesoru. Postup je takový, že je programem mikroprocesoru alokován kontinuální blok paměti a do FPGA je předána jeho adresa. Následně je možné pomocí AXI komunikace daný blok paměti přečíst přímo z FPGA, nebo do něj data zapsat. K obsluze těchto přenosů slouží právě IP DMA. To také umožňuje po dokončení přenosu vyvolat přerušení, aby mohl procesor okamžitě jednat (například zahájit další přenos). Kromě samotného přenosu dat mezi FPGA a pamětí se totiž DMA chová jako slave. Zahájení přenosu (kterýmkoli směrem) musí vždy iniciovat mikroprocesor pomocí doplňkové AXI komunikace (přes ní posílá i adresu datového pole v paměti). Schéma zapojení DMA je znázorněno na obrázku 7.10.

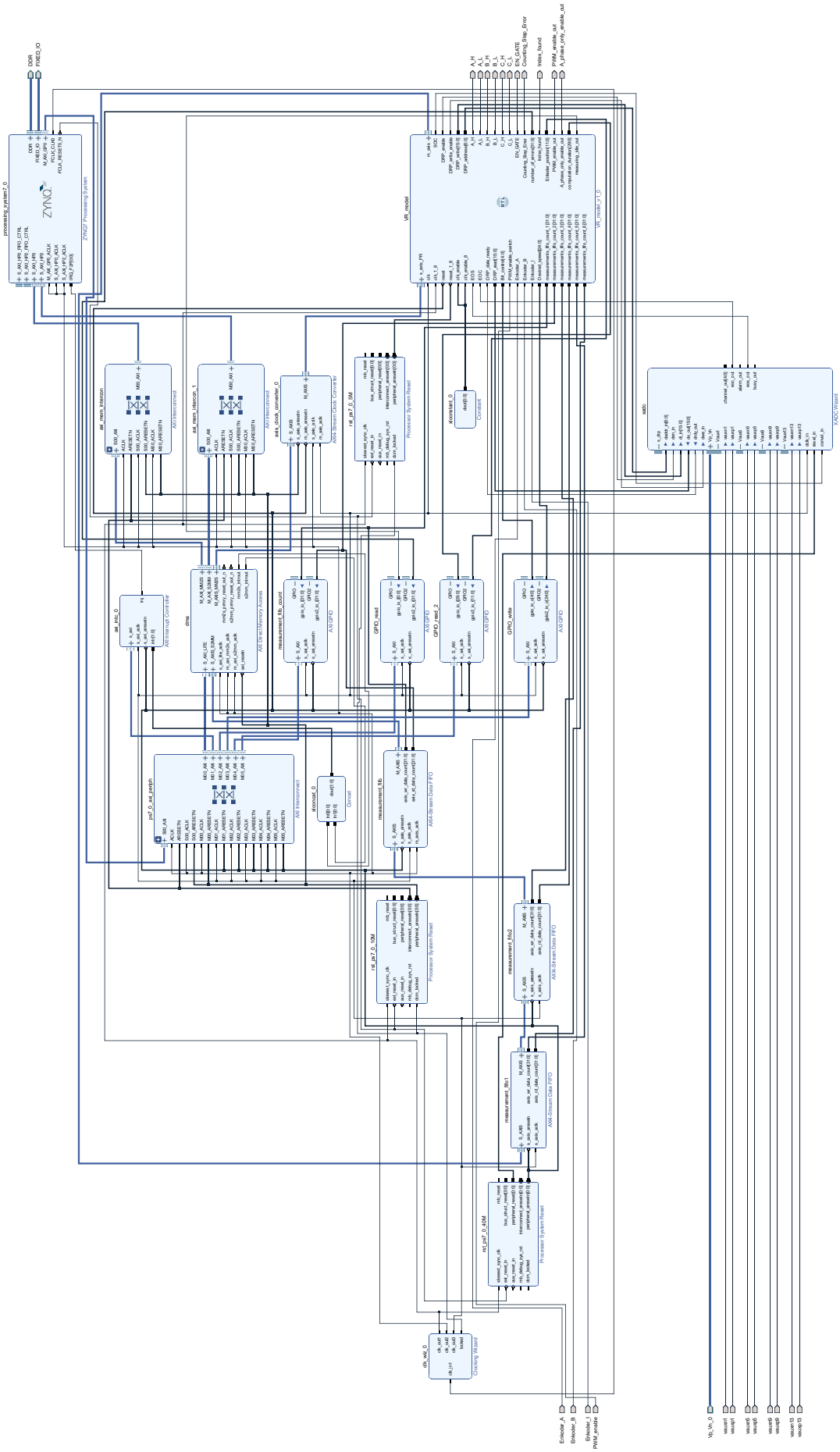


Obr. 7.10: Schéma zapojení AXI DMA z dokumentace PYNQ. Overlay je v této Python knihovně označení pro konfiguraci FPGA. PS je označení pro procesorový systém. *M* označuje *master* komunikace, *S* naopak *slave*. [50]

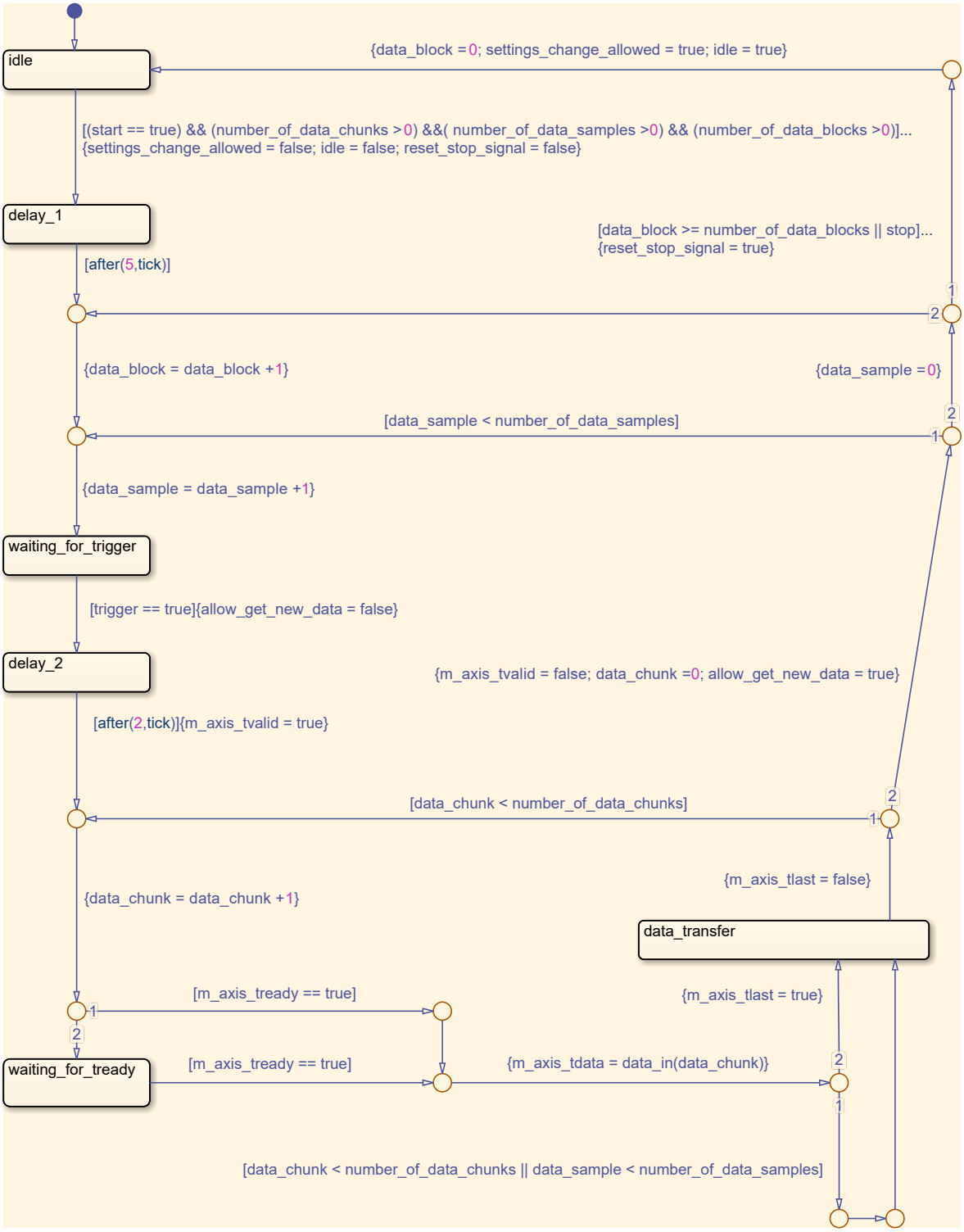
AXI protokol existuje ve třech verzích [47] – standardní, *Lite* a *Stream*. Standardní obsahuje velké množství různých signálů a má rozsáhlou specifikaci, Lite je jeho odlehčenou verzí. Nejjednodušší je však Stream, který má jen malé množství řídicích signálů a používá se pro jednosměrný přenos dat mezi jednotlivými IP v rámci FPGA. Použité IP DMA využívá všechny tři varianty [48] – standardní pro přenos dat do paměti procesoru a opačně, Lite pro komunikaci s procesorem a Stream pro přenos dat ze zdrojového, respektive do cílového IP. Tím je v tomto případě RTL modul generovaný ze Simulinku, pro využití DMA tak muselo být na základě specifikace [49] vytvořeno rozhraní pracující s AXI Stream protokolem.

V realizovaném algoritmu je DMA využito ke dvěma věcem – nastavení většiny parametrů algoritmu (parametry regulátorů, frekvence PWM, parametry měření apod.) a záznam měřených hodnot. Nastavení parametrů je přenos směrem z paměti do FPGA a jsou posílány vždy všechny parametry. Přenos iniciuje uživatel po spuštění algoritmu, nebo pokud potřebuje některý parametr změnit. Hodnoty přijaté pomocí AXI Streamu jsou pomocí stavového stroje uloženy do pole registrů, odkud jsou rozvedeny do celého algoritmu (vždy se čeká na přijetí všech hodnot).

Záznam měřených hodnot je komplikovanější. Je to z několika důvodů. Aby bylo možné data průběžně zobrazovat, musí být rozdělena do určitých bloků (*data_block*) a odesílána průběžně. Průběžné odesílání je komplikované, protože každý jednotlivý DMA přenos musí být zahájen procesorem. Tam však na rozdíl od FPGA nelze jednoduše zaručit maximální latenci (alespoň ne v Pythonu v kombinaci se způsobem, jakým knihovna Pynq nakládá s přerušeními). Z tohoto důvodu je nutné začlenit frontu (FIFO). K tomu je ve Vivado k dispozici IP *AXI4-Stream Data FIFO*. To zajistí efektivní implementaci fronty dle nastavených parametrů. Maximální délka fronty v tomto IP je 32 768, lze jich však zařadit více za sebe. Na použitém hardwaru jsou bloky RAM (BRAM) konfigurovatelné jako FIFO, což z nich dělá ideální kandidáty. Po dokončení návrhu celého řídicího algoritmu tak byl počet front zvolen podle



Obr. 7.11: Schéma zapojení ve Vivado



Obr. 7.12: Převod měřených dat na AXI4 Stream pomocí stavového diagramu. V porovnání se stavovým diagramem na obrázku 7.2 je zde méně stavů, ale složitější (rozvětvené) přechody. Akce jsou v tomto případě prováděny pouze při přechodu ze stavu do stavu (potenciálně i do stejného) a jsou zapsány ve složených závorkách. Podmínky přechodu jsou opět zapsány v hranatých závorkách. Při rozvětvení přechodu (žluté kružnice) je pořadí kontroly podmínek uvedeno u jednotlivých větví čísly. Pokud jsou podmínky dané větve splněny, ostatní větve se nevyhodnocují. Dílčí části přechodů se z hlediska výpočtů vyhodnocují postupně. Při přechodu ze stavu *data_transfer* do sebe sama je tedy hodnota *data_chunk* nejdříve porovnána s číslem *number_of_data_chunks*, poté je teprve navýšena o 1 a nová hodnota je nakonec použita jako index pole *data_in*.

zbývajícího dostupného množství těchto bloků. Celkem tedy byly použity 3 fronty, každá o maximální nastavitelné délce. Každá z nich je tvořena 37 BRAM (celkem je dostupných 140 BRAM). Schéma celého zapojení je zobrazeno na obrázku 7.11.

Vzhledem k tomu, že všechna data musí být ve stejném datovém formátu, bylo nutné je nejprve převést (totéž platí i o parametrech algoritmu přijatých z DMA). Přes DMA je možné přenášet čísla s bitovou délkou v mocninách 2, a to od 8 do 1024. K přenosu měření byl zvolen 32bitový celočíselný formát bez znaménka (pro přenos parametrů 64bitový celočíselný se znaménkem). Čísla tím často ztratí svou hodnotu, je proto nutné provádět převody tak, aby bylo možné ji (v Pythonu) z výsledné bitové reprezentace převodem znovu získat. V případě kratších formátů bez znaménka lze jednoduše doplnit nuly na straně MSB (nejvýznamnější bit) a považovat je za celočíselné (beze změny hodnot dané série bitů, pomocí které bylo číslo původě realizováno), nebo dokonce pro úsporu spojit několik hodnot za sebe. Větší čísla je nutné rozdělit (to se týká pouze časové značky). Formáty se znaménkem kratší než 32 bitů jsou rozšířeny na 32 bitů beze změny reprezentované hodnoty a následně považovány za celočíselný formát bez znaménka (beze změny hodnot jakýchkoli bitů).

Vytvoření AXI Streamu z naměřených dat bylo realizováno pomocí stavového stroje, viz obrázek 7.12. V každé periodě PWM dochází k měření proudů a výpočtům regulátorů. Všechna zaznamenávaná data z jedné periody PWM tvoří jeden vzorek (*data_sample*). Jednotlivé kusy dat (32 bitů) jsou zde označovány jako *data_chunk*. Jejich počet v jednom vzorku závisí na nastavení měření. Odesílají se pouze kusy dat obsahující veličiny zvolené uživatelem (toto nastavení není možné v průběhu měření měnit). V každém DMA přenosu je možné odeslat více vzorků (jeden *data_block*). Jejich počet opět volí uživatel pro celé měření.

Měření může uživatel zahájit a ukončit prostřednictvím uživatelského rozhraní. Tato informace je do FPGA předána pomocí AXI GPIO. Měření je také možné omezit maximálním počtem bloků dat. Záznam dat má uživatel možnost ovlivnit ještě jedním parametrem. Zaznamenávat totiž nemusí každý vzorek (každou periodu PWM), ale třeba každý druhý nebo třetí. Toto číslo je možné volit prakticky libovolně (nesmí to být nula a musí se vejít do 32bitové proměnné – maximální hodnota $2^{32} - 1$, jejíž překročení nemá praktický význam).

Pokud by došlo k naplnění všech front (nevhodně nastavené parametry měření – malá velikost bloků dat a vysoká frekvence zaznamenávaných vzorků – přílišná režie, která je v Pythonu pomalá), nedojde k uložení nových vzorků, dokud není aktuální vzorek celý odeslán. I pokud by tedy došlo k výpadku vzorků, žádný odeslaný vzorek nebude porušený. Každý vzorek navíc obsahuje časovou značku. Byl také implementován čítač zmeškaných vzorků, který může uživatel kdykoli přečíst a mezi jednotlivými měřeními resetovat.

Nakonec je nutné umožnit procesoru zjistit, že byl dokončen poslední DMA přenos daného měření. K tomu byl vytvořen informační bit, který je možné přečíst pomocí AXI GPIO. Ten je nastaven na hodnotu jedna, pokud stavový stroj zajišťující převod na AXI Stream již předal poslední vzorek z daného měření a zároveň jsou všechny fronty prázdné. Tento bit je kontrolován před každým spuštěním nového přenosu. Zahájený přenos totiž není možné pomocí knihovny Pynq přerušit. Pokud jsou pro nové měření změněny parametry, může být délka bloku dat odesílaných v rámci jednoho přenosu jiná. V takovém případě je nutné alokovat nové pole v paměti, které bude této délce odpovídat.

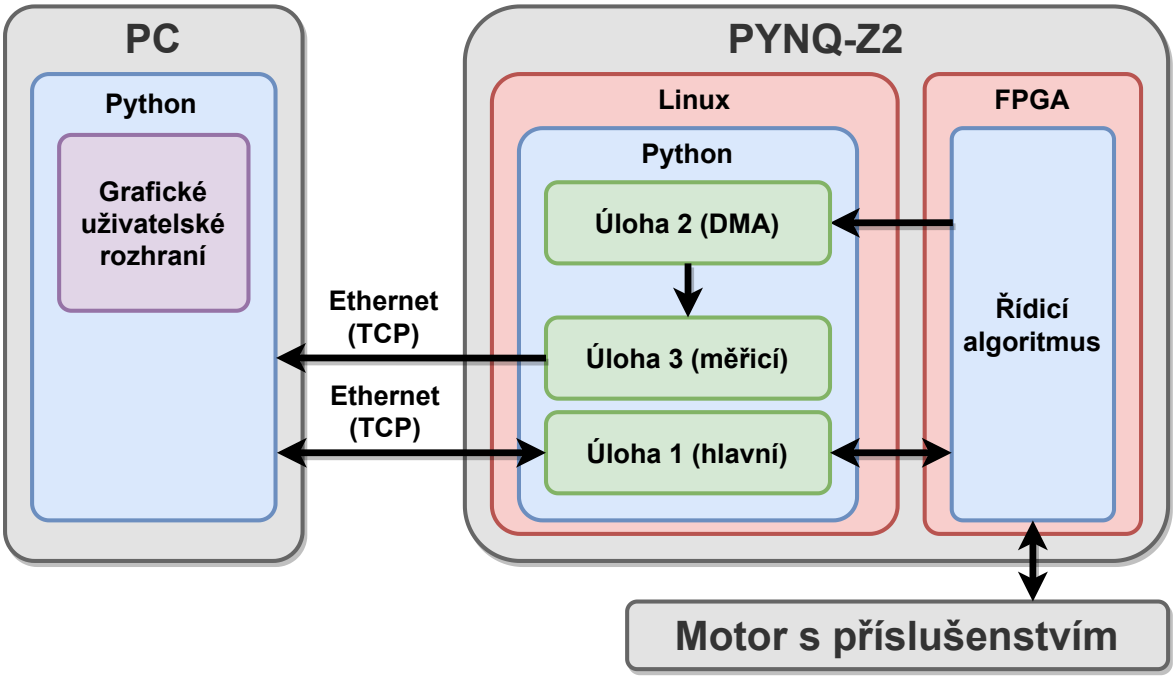
8 Realizace uživatelského rozhraní

Základním způsobem programování procesorové části čipu na vývojové desce PYNQ-Z2 je využití softwaru Jupyter Notebook, nebo jeho nástupce Jupyter Lab. Tyto nástroje vytvoří na PYNQ-Z2 webový server, ke kterému je následně možné přistupovat pomocí webového prohlížeče. V obou případech je možné vytvářet interaktivní dokumenty označované jako *Notebook* (.ipynb). Při jejich otevření je přímo na PYNQ-Z2 spuštěno jádro interaktivního Pythonu. Na rozdíl od klasických skriptů Pythonu, kde je po vykonání zadaných příkazů proces interpretru ukončen, zde zůstává spuštěný až do manuálního vypnutí/restartu uživatelem. Uživatel tak může vytvářet buňky s příkazy a jednotlivě je spouštět (i opakovaně a v libovolném pořadí). V kombinaci s předinstalovanou knihovnou Pynq tak lze jednoduše nahrát konfiguraci do FPGA a ovládat jednotlivá IP připojená k procesorovému systému přes AXI komunikaci.

Tento postup je užitečný pro rychlé otestování navrhovaných řešení, jako uživatelské rozhraní pro navrhovanou aplikaci však není úplně vhodný. Místo toho byla využita možnost spustit prostřednictvím linuxového terminálu klasický (neinteraktivní) Python proces s vlastním programem, který také může využívat knihovnu Pynq. Uživatelské rozhraní bylo vytvořeno v Pythonu pomocí knihovny *Flet* [51] a je spouštěno na PC. Na desce PYNQ-Z2 byl v Pythonu vytvořen TCP server, který přijímá pokyny z programu spuštěného na PC a odesílá mu naměřená data. Tímto způsobem je zajištěno přívětivější ovládání a lepší výkon (zpracování dat je prováděno až na PC). Architektura celé aplikace je znázorněna na obrázku 8.1.

Knihovna Flet byla vybrána kvůli kombinaci moderního vzhledu, podpory grafů, způsobu tvorby GUI (grafické uživatelské rozhraní) a dobré podpory konceptu programování *asynchronní vstup/výstup*. Ten je v Pythonu realizován pomocí standardní knihovny *asincio* [52]. Jedná se o způsob souběžného programování, který je vhodný pro aplikace, kde je nutné provádět více úloh současně, ale jednotlivé úlohy často dlouze čekají na vstupy nebo výstupy (například pokyn od uživatele, načtení/odeslání dat přes internet apod.). To je přesně případ obou částí této aplikace (klient na PC i server na PYNQ-Z2). V porovnání s ostatními způsoby souběžného programování, které Python nabízí (vícevláknové a víceprocesové), je v tomto případě asynchronní I/O efektivnější i jednodušší. Umožňuje totiž programátorovi zvolit, kde bude docházet k přepnutí mezi úlohami. Není tedy nutné ošetřovat případy, kdy operační systém v nevhodnou dobu přepne mezi úlohami, které pracují se stejnými daty. Navíc program nikdy není zdržován vykonáváním úlohy, která zrovna na něco čeká, pokud je jiná úloha připravena k činnosti.

Pro usnadnění komunikace mezi klientem a serverem byla komunikace rozdělena na dvě části, z nichž každá má navázané vlastní TCP spojení. Hlavní komunikace (bude



Obr. 8.1: Architektura vytvořené aplikace. Na straně klienta (PC) nejsou asynchronní úlohy zobrazeny z důvodu jejich velkého množství.

označována také jako *řídicí*) zajišťuje vše kromě přenosu měřených dat přijatých z FPGA přes DMA. Ten zajišťuje druhá komunikace (bude označována jako *měřicí*). Hlavní komunikace probíhá synchronním způsobem. Klient pošle požadavek a čeká na odpověď (myšleno na aplikační úrovni, provedení samotného odeslání/přijetí dat přes TCP protokol a s tím související nízkoúrovňovou komunikaci zajišťuje operační systém). Měřicí komunikace probíhá jednosměrně, server průběžně odesílá naměřená data a neočekává žádnou odpověď.

Obě komunikace mají vytvořené své vlastní komunikační protokoly (postavené nad použitým TCP protokolem). Řídicí komunikace obsahuje ve směru od klienta k serveru 4bajtovou hlavičku, která obsahuje nejdříve 2bajtový číselný kód požadované operace a pak 2bajtovou délku dat. Za ní následují případná data (může být i nulová délka dat, například požadavek na zahájení řízení žádná data nepotřebuje). Uvedením délky dat v hlavičce odpadá nutnost ukončovacího znaku, takže nemusí být prováděna zbytečná konverze dat na text. Odpověď má stejnou strukturu, ale hlavička má pouze 2 bajty. Číselný kód odpovědi i informace o délce dat jsou jednobajtová čísla. Měřicí komunikace má v hlavičce pouze 8bajtovou délku dat, kód zde není potřeba. Informace o dokončení měření se posílá formou dodatečné zprávy s nulovou délkou dat.

Program serveru na PYNQ-Z2 se skládá z několika asynchronních úloh:

- Hlavní úloha (komunikace (řídicí) s klientem a provádění jeho pokynů, spouštění ostatních úloh)

- Úloha obstarávající DMA (pouze směr z FPGA do Pythonu – záznam měřených dat)
- Úloha odesílající měřená data klientovi (měřicí komunikace)

Podstatné zde je, že knihovna Pynq podporuje knihovnu asincio. Úloha obstarávající DMA tak může po zahájení přenosu a odevzdání dat z předchozího přenosu předat řízení ostatním úlohám a je obnovena až po dokončení přenosu. K tomu je využito přerušení z FPGA vyvolané IP DMA. To však nevyvolá okamžité obnovení úlohy. Zachycení přerušení zajistí knihovna sama a úloze je řízení předáno ve chvíli, kdy se jiná, aktuálně probíhající úloha vzdá řízení. Okamžité obnovení úlohy je provedeno, pokud všechny ostatní úlohy také na něco čekají a žádná tedy nemá řízení programu. Ukázka části programu úlohy obstarávající DMA je zobrazena na obrázku 8.2.

Na straně klienta je úloh mnohem více. Hlavní úloha nejdříve vytvoří grafické uživatelské rozhraní. Při vytváření jsou tlačítkům přiřazeny úlohy. Na základě pokynu uživatele tak probíhají další akce. Zároveň jsou při připojování k serveru vytvořeny úlohy pro obsluhu měřicí komunikace, zpracovávání dat a obsluhu grafů.

8.1 Ovládání aplikace

Po spuštění programů serveru i klienta (viz kapitola 10) je většina možností nedostupná. Lze změnit jazyk uživatelského rozhraní (výchozí je čeština, druhou možností je angličtina), načíst uložená nastavení a připojit se k serveru. Pro úspěšné načtení nastavení se musí soubor s uloženým nastavením nacházet ve složce, ve které byl spuštěn Python program klienta (nemusí se jednat o stejnou složku, ve které je program uložen). Pro připojení je nutné zadat IP adresu (IPv4) serveru a porty obou komunikací. Port 1 je pro řídicí komunikaci, port 2 pro měřicí komunikaci (lze zjistit najetím myší nad textové pole). Porty jsou nastaveny v hlavním souboru programu serveru *main.py* a vypsány i s IP adresou v terminálu, ve kterém byl server spuštěn.

Po připojení k serveru se zpřístupní veškerá nastavení, která se do FPGA nahrávají přes DMA. Jejich nahrání do FPGA je iniciováno tlačítkem *Použít nastavení* a jsou vždy nahrány všechny parametry. Obecně je nutné v celé aplikaci akce vyvolávat zmáčknutím tlačítek, ostatní prvky akce nevyvolávají.

Sekce parametrů je rozdělena do několika částí. V první části si uživatel může navolit, které veličiny chce zaznamenávat a zobrazovat. Jedná se o

- žádané a měřené proudy i_d a i_q,
- výstupy proudových regulátorů,
- fázové proudy,

```
while True:
    ## allow control takeover
    await asyncio.sleep(0)

    ## await odd transfer (recv_buff_1)
    await overlay.dma_receive_wait_async()

    ## check, if next transfer is necessary
    ## (don't start transfer if measuring is completed and fifo is empty)
    if overlay.measuring_idle():
        await queue.put(recv_buff_1.buffer.tobytes())

        ## indicate end of measuring
        await queue.put(None)

        ## idle
        # await/check this when canceling task
        # check it when starting next measurement
        event_recv_handling_idle.set()
        event_recv_handling_not_idle.clear()

        ## wake
        await event_recv_handling_wake.wait()
        event_recv_handling_idle.clear()
        event_recv_handling_wake.clear()
        if event_recv_handling_stop.is_set():
            break

        overlay.dma_receive_transfer(recv_buff_2)
        event_recv_handling_not_idle.set() # 'start measuring' awaits this event
    else:
        overlay.dma_receive_transfer(recv_buff_2)
        await queue.put(recv_buff_1.buffer.tobytes())
```

Obr. 8.2: Ukázka části asynchronního kódu úlohy obstarávající DMA (nejedná se o celou úlohu). Klíčové slovo *await* je před funkcemi, u kterých je možné, že nebudou provedeny okamžitě. Pokud nejsou provedeny okamžitě (na něco čekají), dojde k předání řízení smyčce událostí, která zvolí další úlohu, která dostane řízení. Na zobrazeném kódu úloha čeká na dokončení aktuálního DMA přenosu a poté kontroluje, zda už byl poslední. Pokud poslední nebyl, zahájí další přenos do druhého alokovaného pole a následně předá data z původního přenosu do fronty, odkud si je může převzít jiná úloha. V rámci *while* smyčky je ještě jednou zopakován celý zobrazený kód, ale s prohozenými datovými poli. Pokud byl DMA přenos už poslední, dojde k odevzdání dat (bez zahajování dalšího přenosu) a upozornění na to, že byla poslední. Úloha pak čeká na probuzení hlavní úlohou. Před while smyčkou jsou ještě další příkazy provedené při vytvoření úlohy, včetně zahájení prvního přenosu.

- napětí zdroje,
- žádanou a měřenou rychlost,
- regulační odchylku rychlosti,
- pozici enkodéru a úhel θ,
- režim PWM (zda je zapnutá regulace či se měří poloha indexu enkodéru),
- informaci o tom, zda je PWM (řízení) zapnuto.

Všechny hodnoty jsou záměrně brány z FPGA. I ty, které nastavuje uživatel, jako třeba žádanou rychlost. Je to z toho důvodu, aby v zaznamenávaných datech seděly na vzorek přesně hodnoty, které řízení skutečně využívá. Spuštění PWM je navíc z bezpečnostních důvodů řízeno logickým součinem (musí být oboje aktivní) nastavení z uživatelského rozhraní (*Start*/*Stop*) a hodnoty hardwarového přepínače *SW1* umístěného přímo na vývojové desce. Uložení hodnoty *PWM zapnuto* přímo z uživatelského rozhraní by tak vůbec nemuselo odpovídat. Tato logická proměnná se používá k zapnutí PWM výstupů z FPGA (jinak jsou hodnoty všech 6 řídicích signálů na nule) a je i vyvedena jako digitální výstup do výkonového modulu. Ten vyžaduje její pozitivní hodnotu, jinak odpojí všechny fáze motoru od napětí i nuly zdroje (všech 6 výkonových tranzistorů je vypnutých). Tuto hodnotu, stejně tak jako informaci o režimu řízení, je možné zaznamenávat, aby byl stav řízení jasný i z měřených hodnot při jejich dodatečné analýze. Zároveň je hodnota *PWM zapnuto* indikována LED diodou *LD2* (rozsvícená značí zapnuto).

Za nastavením měřených veličin je k dispozici volba frekvence PWM. Zde jsou na výběr podporované frekvence. Dále následuje výběr režimu řízení. Volba *Pouze fáze A* umožňuje nastavit uživateli procentuální hodnotu napětí na fázi A, přičemž ostatní fáze jsou ponechány na nule. To zajistí natočení motoru do polohy, ve které je možné přímo odečíst úhel θ_{offset} (úhel mezi indexem enkodéru a rotorovým souřadným systémem *d*-*q*), který je pro řízení nezbytný. Dále je možné obrátit směr vyhodnocování enkodéru, pokud není v souladu se zapojením fází (prohození 2 fází změní směr otáčení motoru). Pokud byl změněn směr otáčení, je nutné znovu určit úhel θ_{offset}. Nakonec je možné přepnout mezi proudovou a rychlostní regulací a zvolit, jestli mají být parametry proudových regulátorů přepočítávány podle měřeného napětí.

Následují číselné parametry. Ty jsou opět rozděleny do kategorií. Navíc se zobrazují vždy jen ty, které jsou relevantní pro daný režim řízení. Nejdříve jsou parametry měření. Je možné zvolit počet vzorků dat v jednom bloku n_S, maximální počet bloků v jednom měření (měření lze ukončit i dříve) a předdělič zaznamenávaných vzorků n_{SP}. Tyto parametry byly popsány v kapitole 7.6 a posílají se do FPGA společně s ostatními parametry. Dále jsou k dispozici parametry předdělič zobrazovaných vzorků

n_{SPG} a počet bloků dat v grafu n_{BG}. Tyto parametry slouží k nastavení průběžného vykreslování měřených hodnot. V grafu je zobrazeno vždy zadané množství nejnovějších bloků dat, přičemž z každého bloku je zobrazen pouze každý n_{SPG}-tý vzorek. Poměr $\frac{n_S}{n_{SPG}}$ musí být vždy celé číslo. Frekvenci získávání nových bloků dat je možné určit výpočtem

$$f_{DT} = \frac{f_{PWM}}{n_S \cdot n_{SP}} \tag{8.1}$$

kde f_{PWM} je frekvence PWM. Počet zobrazovaných vzorků na každou sekundu lze stanovit vztahem

$$n_{GSPS} = \frac{f_{PWM}}{n_{SP} \cdot n_{SPG}} \tag{8.2}$$

a délku dat zobrazovaných v grafu (v sekundách)

$$T_G = \frac{n_{BG}}{f_{DT}} = \frac{n_{BG} \cdot n_S \cdot n_{SP}}{f_{PWM}} \tag{8.3}$$

Dalšími číselnými parametry jsou parametry proudových regulátorů. Ty jsou přepočítány tak, aby zesílení jednotlivých složek číselně odpovídalo zesílení z vektoru proudů v ampérech na vektor napětí ve voltech. Je zde kompenzován normalizační koeficient $\frac{2}{3}$ zavedený v Clarkeové transformaci i zmenšení vektoru napětí vektorovou modulací. Zároveň je v integrační složce kompenzován vliv vzorkovací frekvence, viz rovnice (7.5). Pokud je vypnuto měření napětí zdroje, je pro toto napětí k dispozici parametr. Hodnoty omezení výstupů proudových regulátorů přepočítány na volty nejsou, zůstávají v rozsahu -1 až 1. Jejich přepočet by neměl praktický význam (při nižším napětí zdroje je zapotřebí větší normalizovaný, amplitudově invariantní vektor napětí). Vektorový součet složek $\vec{v}_d$ a $\vec{v}_q$ (v normalizovaném, amplitudově invariantním stavu) nesmí v absolutní hodnotě přesáhnout hodnotu 1. Navíc zde musí být rezerva, aby bylo možné měřit proudy (vektor o absolutní hodnotě 1 by při některých úhlech způsobil trvalé sepnutí některé fáze na napětí zdroje, což by způsobilo neplatné hodnoty měření na dané fázi).

Parametry rychlostního regulátoru obdobně přepočítávány nejsou. Jediným přepočtem je zde kompenzace vlivu vzorkovací frekvence. Vstupem regulátoru je regulační odchylka rychlosti v otáčkách za minutu a výstupem žádaný proud i_q v rozsahu -1 a 1. Naopak jsou zde na ampéry přepočítány hodnoty omezení výstupu regulátoru.

Mezi číselnými parametry jsou ještě úhel θ_{offset}, žádaná hodnota na fázi A, kalibrační parametry popsané v kapitole 7.1 a žádané hodnoty proudů i_d a i_q. Vždy jen to, co je relevantní pro daný režim.

Po tom, co jsou prvně nahrány parametry regulace do FPGA, je zpřístupněn ovládací panel. Před zahájením řízení je nutné motorem manuálně pootočit, aby byl alespoň jednou načten indexační pulz enkodéru (počáteční poloha motoru není známá). Po

jeho prvním načtení se rozsvítí LED dioda *LD3*. Stejně tak je nutné motorem pootočit a načíst index, pokud byl přepnut směr enkodéru (a to i v případě, že byl přepnut zpět a mezitím došlo k sebemenšímu pootočení motoru). Z tohoto důvodu je také k dispozici tlačítko, které resetuje registr indexu enkodéru a LED dioda zhasne.

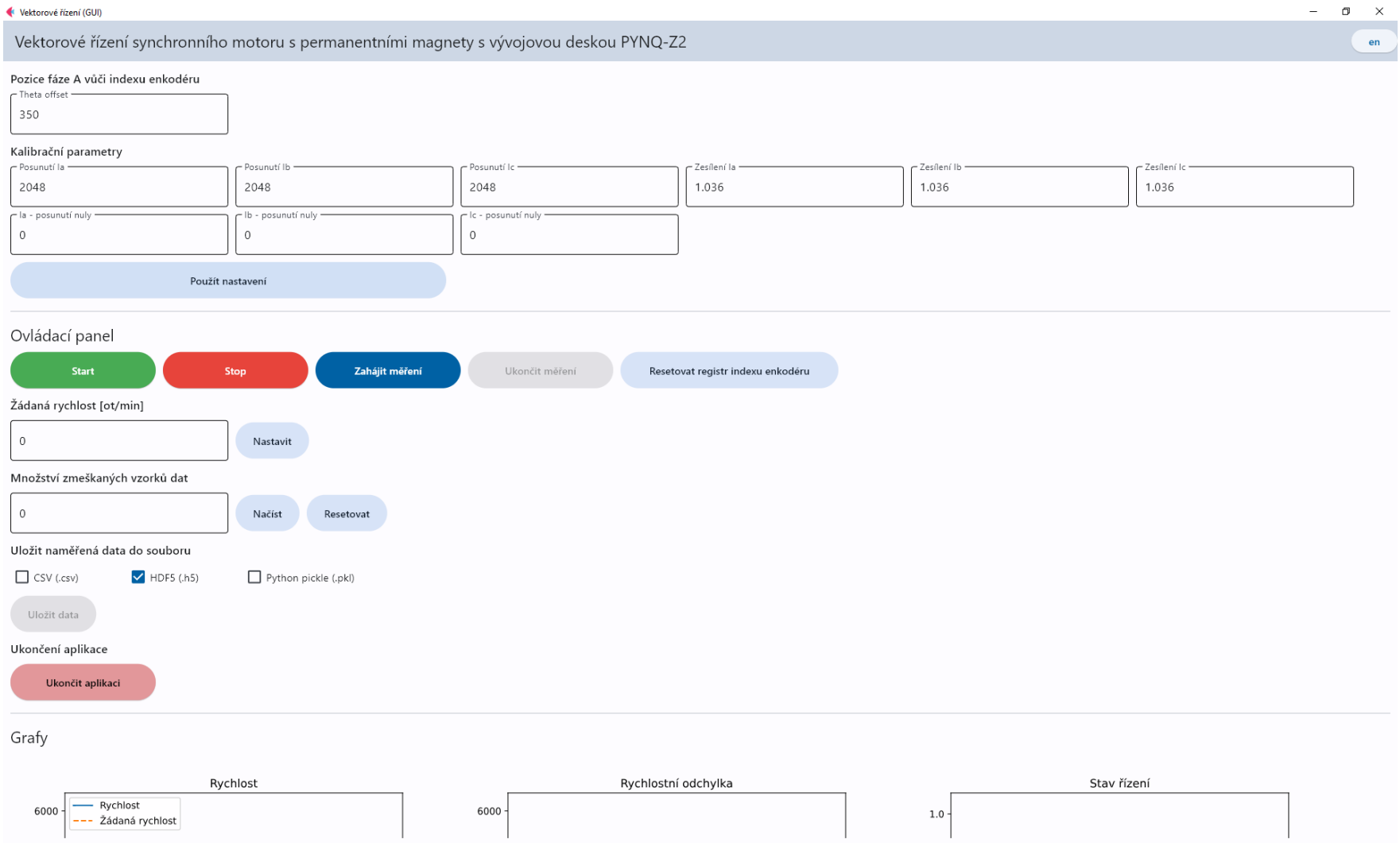
Na ovládacím panelu jsou tlačítka pro spuštění řízení (povolení PWM) (*Start*) a jeho vypnutí (*Stop*). Také jsou zde tlačítka pro zahájení a ukončení měření. Tlačítko pro zahájení měření zablokuje prvky uživatelského rozhraní (tzv. zašednou) nastavující parametry, které není možné během měření měnit. Zároveň, aby nedošlo k znepřístupnění těch, které naopak mají být k dispozici, obnoví v uživatelském rozhraní hodnoty a nastavení, které byly naposledy poslány do FPGA přes DMA. Jinak by totiž mohla být příslušná textová pole schována, pokud uživatel mezitím změnil režim měření (např. z proudové na rychlostní regulaci) a nenahrál je do FPGA (*Použít nastavení*).

Dále jsou zde textová pole a tlačítka sloužící k nastavení žádané rychlosti, nastavení napětí na fázi A a načtení a resetování počtu zmeškaných vzorků. Opět pouze ta, která jsou pro daný režim relevantní. Resetování počtu zmeškaných vzorků není možné v průběhu měření. Akce na ovládacím panelu jsou do FPGA přenášeny pomocí AXI GPIO, viz kapitola 7.6.

Nakonec je na ovládacím panelu možnost uložit naměřená data v uživatelem zvoleném formátu (CSV, HDF5 a Python pickle) a ukončit aplikaci. Uložení naměřených hodnot je možné jen z posledního měření.

Za ovládacím panelem následují grafy. Které z nich budou zobrazeny záleží na naposledy nahraných parametrech (do FPGA). Grafy se průběžně aktualizují během měření. Úloha, která má aktualizaci grafů na starosti, provede vždy aktualizaci všech hodnot v grafech, a pak asynchronně žádá knihovnu Flet, aby aktualizovala grafy v uživatelském rozhraní. Žádost o aktualizaci grafů je podávána jednotlivě pro každý graf (jen ty aktuálně zobrazené). U grafů je totiž nastaveno, aby nedocházelo k jejich opětovnému vykreslování při překreslování zbytku uživatelského rozhraní, což zrychluje odezvu na ostatní akce. Po dokončení aktualizace a obnovení všech zobrazovaných grafů úloha 250 ms čeká, než začne znovu. Aktualizace grafu tak probíhá na frekvenci necelých 4 Hz.

Celé uživatelské rozhraní je vytvořeno jako jedno okno. Rozhraní se automaticky přizpůsobuje šířce okna a je možné v něm vertikálně rolovat. Přizpůsobení šířce okna je provedeno zalamováním řádků prvků. Každá skupina prvků, které jsou vedle sebe, je vytvořena jako řádek s možností zalomení. Prvky, které se již celé nevejdou na daný řádek, jsou posunuty na nový. Jednu řadu tvoří například veškeré grafy nebo veškeré číselné parametry proudových regulátorů. Ukázka části uživatelského rozhraní je na obrázku 8.3, další ukázky jsou v příloze.



Obr. 8.3: Ukázka části uživatelského rozhraní

9 Otestování a zhodnocení vytvořené aplikace

Vytvořená aplikace byla otestována proměřením přechodových charakteristik rychlosti a proudu. Tyto charakteristiky jsou znázorněny na obrázcích 9.1 a 9.2. Obě charakteristiky byly proměřeny s frekvencí PWM 100 kHz a v obou případech byl zaznamenáván každý vzorek. Proměřování přechodové charakteristiky proudu bylo provedeno (v rámci možností) v zablokovaném stavu motoru. Za tímto účelem byl rotor motoru pevně zachycen ve svěráku, stator byl držen ručně. Není tedy možné vyloučit mírné pootočení motoru v průběhu měření, například vlivem rychle vzniklého točivého momentu. To by mohlo být důvodem mírného propadu měřené hodnoty těsně před dosažením žádané hodnoty (v čase přibližně 7 ms od změny žádané hodnoty).

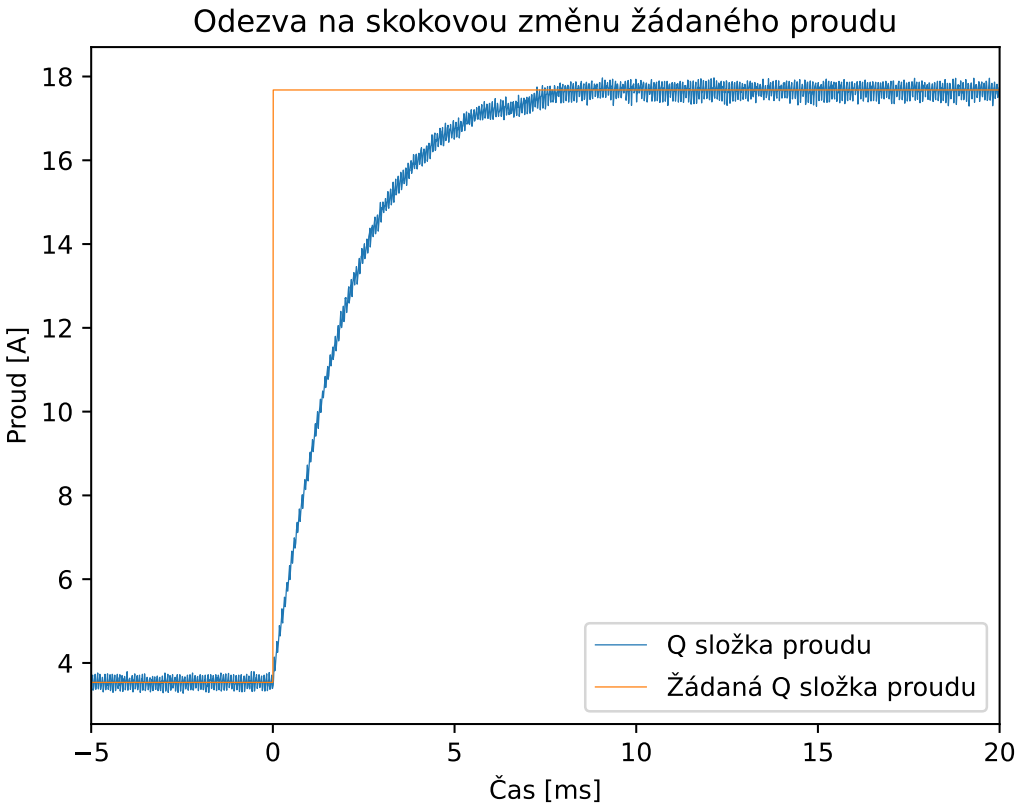
Proměření přechodové charakteristiky rychlosti, zobrazené na obrázku 9.2, bylo provedeno v téměř nezatíženém stavu. Motor byl pouze spojkou připojen k druhému totožnému motoru, ten byl však v tomto případě odpojený od napájení. Rychlostní regulátor byl pro tento případ experimentálně naladěn na relativně agresivní chování. Zároveň ale tak, aby za prvním překmitem již nedošlo k podkmitu.

Řízení se úspěšně podařilo realizovat na nízkonákladové verzi FPGA (čip ZYNQ XC7Z020-1CLG400C na vývojové desce PYNQ-Z2). Přitom stále zůstala rezerva v počtu využitých prvků, jak je patrné z tabulky 9.1 a obrázku 9.3. Zároveň se podařilo realizovat vysokofrekvenční záznam veličin použitých při řízení. Díky této kombinaci je možné realizované řízení využít například jako výchozí bod pro testování různých způsobů filtrace měřených proudů nebo pro testování nových typů proudových regulátorů.

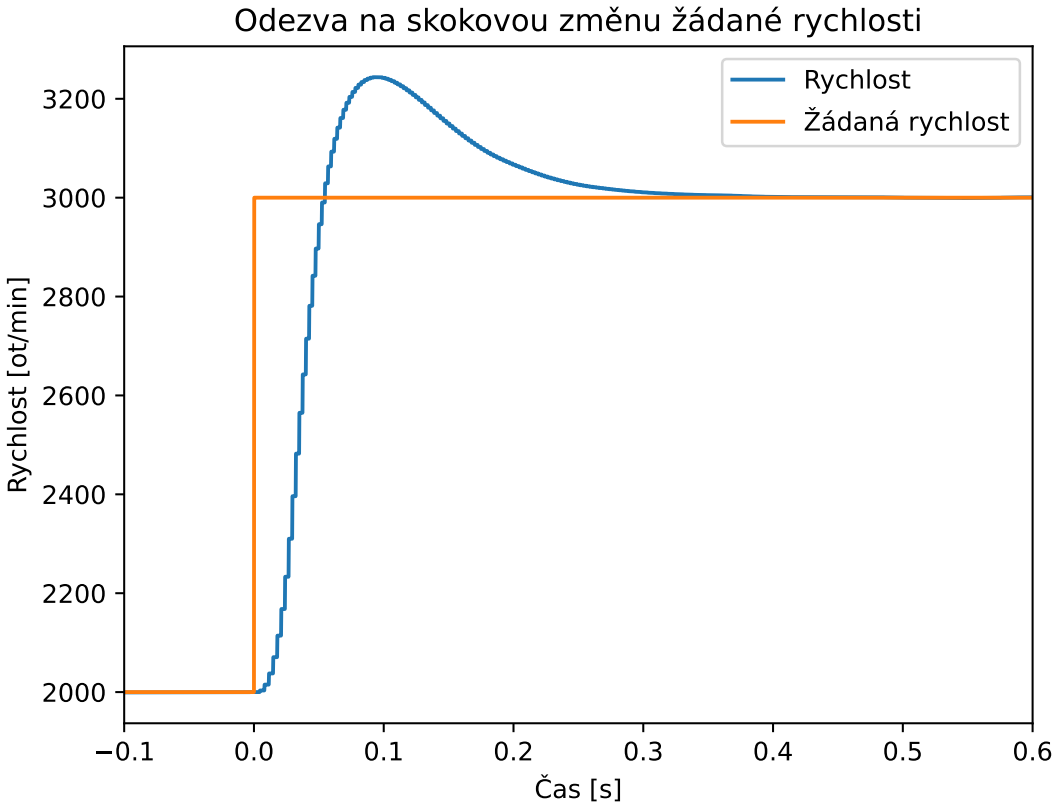
Vysoké využití BRAM je způsobeno záměrně přidáním velké kapacity fronty pro měřená data, viz kapitola 7.6. Vzhledem k tomu, jakým způsobem byla nakonec realizována zbylá část aplikace (mimo FPGA), jedná se o zbytečně velikou rezervu, která však ničemu nevadí. Pokud by tedy při tvorbě pokročilejších algoritmů navazujících na tuto práci bylo zapotřebí větší množství BRAM, než aktuálně zbývá, nebyl by problém nějaké uvolnit zkrácením fronty.

Díky využití konceptu asynchronních vstupů/výstupů bylo dosaženo optimálního využití výkonu procesoru vývojové desky. Vzhledem k tomu, že Python program na desce neprovádí žádné výpočetně náročné operace a při čekání na dokončení přenosů dat (DMA z FPGA a TCP při komunikaci s PC) nezatěžuje procesor, tak při vhodném nastavení parametrů záznamu dat není procesor příliš zatížen. Ukázka využití procesoru a paměti na vývojové desce je zobrazena na obrázku 9.4.

Celková doba měření analogových vstupů (fázové proudy a napětí zdroje) včetně následného čtení výsledků pomocí DRP komunikace byla experimentálně změřena přímo v FPGA. Jedná se o 144 cyklů hlavního hodinového signálu (40 MHz), což



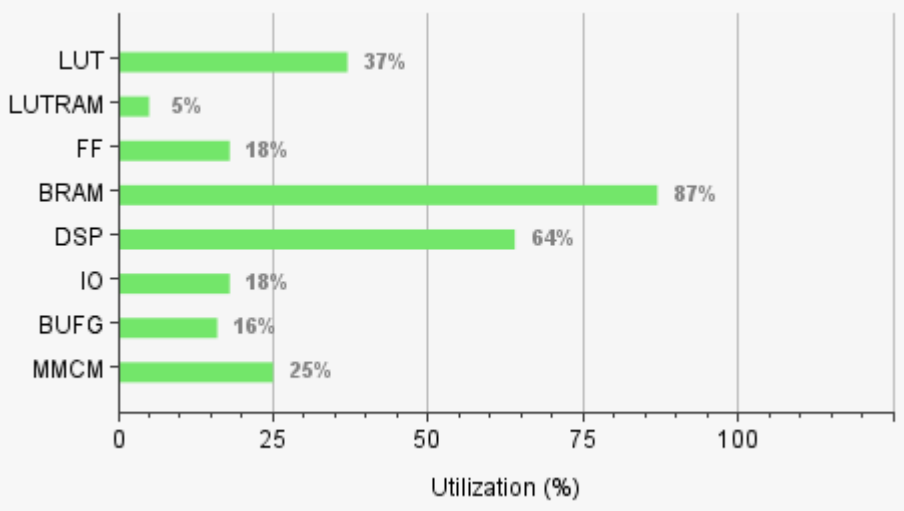
Obr. 9.1: Přechodová charakteristika proudu. Záznam byl proveden na frekvenci 100 kHz.



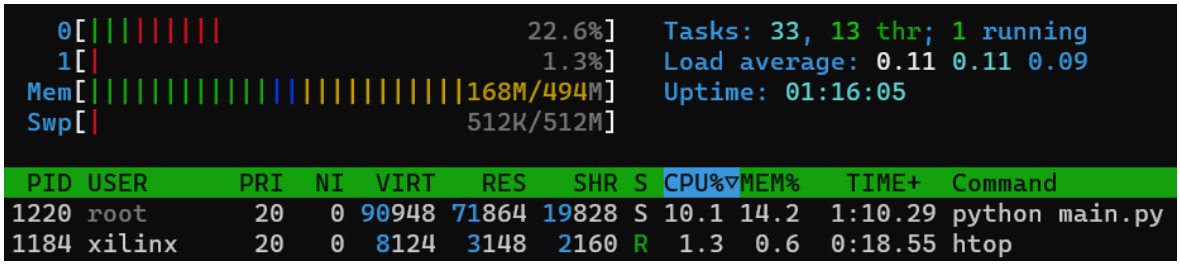
Obr. 9.2: Přechodová charakteristika rychlosti. Záznam byl proveden na frekvenci 100 kHz.

Resource	Utilization	Available	Utilization %
LUT	19863	53200	37.34
LUTRAM	823	17400	4.73
FF	18934	106400	17.80
BRAM	122	140	87.14
DSP	141	220	64.09
IO	23	125	18.40
BUFG	5	32	15.63
MMCM	1	4	25.00

Tab. 9.1: Tabulka zdrojů (prvků/bloků) využitých celým algoritmem (včetně všech IP z Vivado)



Obr. 9.3: Graf procentuálního využití zdrojů celým algoritmem (včetně všech IP z Vivado)



Obr. 9.4: Ukázka vytížení procesoru vývojové desky (příkaz *htop* v linuxovém terminálu). Měření bylo provedeno při frekvenci PWM 100 kHz, přičemž byl zaznamenáván každý vzorek. Vzorky byly přenášeny v blocích po 10 000, což znamená 10 přenosů za vteřinu.

odpovídá 3,6 µs. Následný výpočet trvá 13 cyklů, což odpovídá 325 ns. Celkově tedy 157 cyklů, neboli 3,925 µs. Doba měření se může při některých frekvencích PWM v jednotlivých měřeních mírně lišit (o 1 cyklus), A/D převodník totiž využívá vlastní hodinový signál, který je vytvořen dělením hlavního hodinového signálu dvěma. Záleží tedy, jestli je požadavek na zahájení prvního měření dané sekvence podán na začátku tohoto cyklu nebo až v polovině. Maximální nastavitelná frekvence PWM je 100 kHz, což odpovídá periodě 10 µs.

10 Doplňující informace

Shémata modelu Simulinku a projektu ve Vivado uvedená v této práci slouží primárně pro rychlé získání základní představy o popisovaných částech modelu. Vzhledem k jejich velikosti může být čitelnost omezena, obzvláště v tištěné formě. Zároveň zde nejsou uvedeny zdaleka všechny části modelu. Pro získání lepší představy se čtenáři doporučuje nahlédnout do elektronických příloh, viz kapitola 10.1.

V následujících podkapitolách jsou uvedeny převážně pokyny pro zprovoznění řízení a případné úpravy řídicího algoritmu.

10.1 Simulink

Celý projekt Matlabu, zahrnující hlavně model Simulinku, je k dispozici v příloze. K jeho vytvoření byla použita verze R2022b. Seznam součástí/toolboxů nutných k otevření modelu:

- MATLAB – Version 9.13 (R2022b)
- Simulink – Version 10.6 (R2022b)
- Fixed-Point Designer – Version 7.5 (R2022b)
- Stateflow – Version 10.7 (R2022b)
- DSP System Toolbox – Version 9.15 (R2022b)

Ke generování kódu a dalším akcím mohou být vyžadovány další toolboxy, jako například HDL Coder (Version 4.0, R2022b). Jejich seznam však nebylo možné přesně určit, autor totiž využíval na základě univerzitní licence plnou instalaci (všechny toolboxy byly k dispozici). Seznam závislostí pro otevření lze v projektu Matlabu najít.

Aby bylo možné procházet model i bez instalace Matlabu a potřebných toolboxů, byl vygenerován webový náhled modelu (*webview*). Ten je také k dispozici v příloze. Otevírá se souborem *webview.html* ve webovém prohlížeči, viz struktura příloh. Webové prohlížeče v současné době však blokují některé akce, po otevření se proto zobrazí pokyny k povolení přístupu k souborům, aby mohl být model otevřen (testováno v prohlížeči Google Chrome).

10.2 Vivado

Projekt Vivado byl vytvořen ve verzi 2022.1 dostupné na stránce [53] (k otevření projektu je zapotřebí tato verze). V současné době se jedná o nejstarší verzi, která je přímo zobrazena. Starší verze je možné stáhnout na stejné stránce pod položkou *Vivado Archive*.

V příloze jsou k dispozici soubory potřebné pro vytvoření projektu (ve složce `Vivado_projekt/Vektorove_rizeni_projekt`) a obrázkový návod (včetně označení možností, které mají nebo naopak nemají být vybrány). Také jsou přiloženy soubory vývojové desky, stažené ze stránek výrobce [29], které je nutné před vytvořením projektu přidat do složky s instalací Vivado (dle [54])

`<Xilinx installation directory>/Vivado/<version>/data/boards/board_files`

10.3 PYNQ-Z2

Vývojová deska PYNQ-Z2 potřebuje pro svou funkci micro SD kartu se specifickou distribucí operačního systému. V této práci byla použita verze v3.0.1 dostupná ke stažení na stránkách výrobce [29], nebo v současné době na stránkách projektu [32]. Pokyny k instalaci jsou dostupné v dokumentaci na stránce [55]. Pokyny k zapnutí a používání desky na stránce [56]. V této části je však nutné podotknout, že přímé spojení desky s PC ethernetovým kabelem a připojení přes statickou IP adresu je problematické (někdy vůbec nefunguje). V takovém případě je nutné nastavit na PC (testováno v systému Windows 10 Pro 22H2) sdílení internetového připojení. Ve zmiňovaném systému je nutné jít do nastavení adaptéru, pomocí kterého je počítač připojen k internetu (Ethernet, WI-FI), a nastavit sdílení tohoto připojení do adaptéru, ke kterému je připojena vývojová deska. Systém Windows 10 v tomto případě vytvoří DHCP server a přidělí vývojové desce IP adresu (IPv4). Tu je nutné znát pro spuštění řízení. Druhou možností je připojit PYNQ-Z2 do místní sítě, do které je připojen i počítač (zde je nutné podotknout, že komunikace mezi PC a PYNQ-Z2 není ve většině případů zabezpečena). V obou případech je možné přistupovat k Jupyter Notebooku nejen přes IP adresu, ale i přes *hostname*

```
http://pynq/
```

V případě preference JupyterLabu se jedná o

```
http://pynq/lab
```

Zde je možné přistupovat k terminálu operačního systému vývojové desky a zjistit tak IP adresu příkazem

```
hostname -I
```

Po zjištění IP adresy je nutné nahrát na vývojovou desku potřebné soubory přes Samba protokol, viz opět [56]. Jedná se o soubory s vytvořeným serverem (Python) a soubory `.bit` (bitstream) a `.hwh` získané z Vivado. V souboru *main.py* serveru je nutné upravit IP adresu serveru, aby se shodovala s IP adresou vývojové desky, a cestu k zmíněným `.bit` a `.hwh` souborům. Všechny potřebné soubory jsou k dispozici v příloze ve složce `Python_projekty/DP-Server`. Všechny soubory Pythonu se musí nacházet ve stejném

adresáři (je možné překopírovat celou složku DP-Server z přílohy tak, jak je), stejně tak jako soubory `.bit` a `.hwh` musí být ve stejném adresáři (mohou však být jinde, než soubory Pythonu).

Jakmile je vše připravené, je možné spustit server v terminálu operačního systému vývojové desky. Terminál musí být otevřen přes Jupyter Notebook nebo JupyterLab, aby měl Python přístup ke knihovně Pynq. V případě terminálu otevřeného přes SSH by bylo nutné nejdříve spustit další příkazy. V otevřeném terminálu je nutné se přesunout do složky, ve které jsou soubory serveru nahrány. Spuštění se následně provede příkazem

```
python main.py
```

Spuštění trvá několik vteřin a klient se může připojit, jakmile se v terminálu vypíše informace, že server poslouchá na příslušné adrese. Program je automaticky ukončen, jakmile se klient odpojí.

10.4 Python (klient) – uživatelské rozhraní

Uživatelské rozhraní bylo vytvořeno a testováno v Pythonu 3.10.8 [57] v systému Windows 10 Pro (22H2). Ke správě knihoven byl použit manažer knihoven Poetry [58]. Veškeré potřebné soubory jsou k dispozici v příloze ve složce `Python_projekty/DP-Client`. Po instalaci Pythonu a Poetry je možné vytvořit virtuální prostředí Pythonu a nainstalovat do něj potřebné knihovny jedním příkazem v terminálu. K tomu je nutné zkopírovat zmíněnou složku na místní disk (doporučuje se v celé cestě nepoužívat háčky a čárky, ideálně ani mezery, ačkoli Python by si s tím pravděpodobně poradil) a přesunout se do ní v terminálu. Příkazem

```
poetry install
```

se při dodržení verze Pythonu nainstalují přesně stejné verze závislostí (použitých knihoven a jimi vyžadovaných knihoven). Spuštění uživatelského rozhraní se provede příkazem

```
poetry run python .\src\dp_client\main.py
```

ve stejné složce v terminálu.

11 Závěr

Tato práce se zaměřovala na vytvoření vektorového řízení synchronního motoru s permanentními magnety (PMSM) na vývojové desce PYNQ-Z2. V první části práce byl popsán princip funkce programovatelných hradlových polí (FPGA). Byla zde také provedena rešerše jejich využití v průmyslu a výzkumu, a to převážně v oblasti výkonové elektroniky a elektromotorů. V další části byl popsán způsob programování FPGA, přičemž hlavní pozornost byla věnována softwarům Vivado a Simulink, které byly následně využity při praktické části práce. V té bylo v rámci FPGA (na desce PYNQ-Z2) úspěšně realizováno vektorové řízení. Řízení se zaměřuje na rychlostní regulaci, pro účely ladění však umožňuje i čistě proudovou regulaci. Navíc je také možné s ním určit polohu indexu enkodéru, která je pro řízení nezbytná. Výpočty jsou velmi rychlé, samotný výpočet trvá 325 ns, při započítání doby měření pak celkem 3,925 µs. Napětí je tedy možné modulovat vysokými frekvencemi. Maximum bylo nastaveno na 100 kHz. Bylo také vytvořeno grafické uživatelské rozhraní v programovacím jazyce Python s využitím knihovny Flet. Uživatelské rozhraní je spouštěno na PC a komunikuje s mikroprocesorem vývojové desky přes ethernetové připojení. Ten následně komunikuje s FPGA. Uživatelské rozhraní umožňuje volbu všech parametrů regulace, stejně tak jako měření a zaznamenávání údajů z FPGA (měřené proudy, výstupy regulátorů apod.). Záznam hodnot je možný také na velmi vysokých frekvencích, vždy až do zvolené frekvence modulace napětí. Díky tomu je možné tuto práci využít jako výchozí bod k testování nových algoritmů souvisejících s měřením a regulací proudů. Řízení bylo úspěšně otestováno na motoru Teknic M-2310P-LN-04K. Regulace je stabilní a má rychlou odezvu na změnu žádané hodnoty. Testování rychlostní regulace probíhalo převážně v rychlostech od 1 000 do 4 000 otáček za minutu a při různých hodnotách napětí zdroje. Proudová regulace byla testována v celém rozsahu.

A Struktura příloh

- Prilohy
  - Schema_modelu_webview
    - webview.html
    - ...
  - Matlab_Simulink_projekt
    - VR_projekt.mlproj
  - Vivado_projekt
    - Vektorove_rizeni_projekt
    - Navod
    - board_files
  - Python_projekty
    - DP-Server
    - DP-Client
  - Obrazky

B Seznam použité literatury a zdrojů

[1] MONMASSON, Eric; IDKHAJINE, Lahoucine; CIRSTEA, Marcian N.; BAHRI, Imene; TISAN, Alin; NAOUAR, Mohamed Wissem. FPGAs in Industrial Control Applications. *IEEE Transactions on Industrial Informatics*. 2011, roč. 7, č. 2, s. 224–243. Dostupné z DOI: 10.1109/TII.2011.2123908.

[2] KOLOUCH, Jaromír. *Jazyk Verilog a jeho užití při modelování a syntéze číslicových systémů: příručka*. 1. vyd. Brno: VUTIUM, 2012. ISBN 978-80-214-4516-1.

[3] KASTNER, R.; MATAI, J.; NEUENDORFFER, S. Parallel Programming for FPGAs. *ArXiv e-prints*. 2018. Dostupné z arXiv: 1805.03648.

[4] *Zynq-7000 SoC Data Sheet: Overview*. Online. Xilinx, Inc. DS190 (v1.11.1) July 2, 2018. Dostupné z: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview. [cit. 2023-12-25].

[5] *Versal Architecture and Product Data Sheet: Overview*. Online. Advanced Micro Devices, Inc. DS950 (v1.20) November 14, 2023. Dostupné z: https://docs.xilinx.com/v/u/en-US/ds950-versal-overview. [cit. 2023-12-26].

[6] *AMD Completes Acquisition of Xilinx*. Online. Advanced Micro Devices, Inc. Dostupné z: https://www.amd.com/en/press-releases/2022-02-14-amd-completes-acquisition-xilinx. [cit. 2023-12-26].

[7] *Trademark Information*. Online. Advanced Micro Devices, Inc. Dostupné z: https://www.amd.com/en/legal/trademarks.html. [cit. 2023-12-26].

[8] *Zynq 7000 SoC Technical Reference Manual*. Online. Advanced Micro Devices, Inc. UG585 (v1.14) June 30, 2023. Dostupné z: https://docs.xilinx.com/r/en-US/ug585-zynq-7000-SoC-TRM. [cit. 2023-12-25].

[9] *7-Series Slice Flip Flops*. Online. Xilinx, Inc. Dostupné z: https://www.xilinx.com/video/fpga/7-series-slice-flip-flops.html. [cit. 2023-12-25].

[10] *7-Series Clocking Resources*. Online. Xilinx, Inc. Dostupné z: https://www.xilinx.com/video/fpga/7-series-clocking-resources.html. [cit. 2023-12-25].

[11] MONMASSON, Eric; IDKHAJINE, Lahoucine; NAOUAR, Mohamed Wissem. FPGA-based Controllers. *IEEE Industrial Electronics Magazine*. 2011, roč. 5, č. 1, s. 14–26. Dostupné z DOI: 10.1109/MIE.2011.940250.

[12] MONMASSON, E.; IDKHAJINE, L.; BAHRI, I.; NAOUAR, M-W-; CHARAABI, L. Design methodology and FPGA-based controllers for Power Electronics and drive applications. In: *2010 5th IEEE Conference on Industrial Electronics and Applications*. 2010, s. 2328–2338. Dostupné z DOI: 10.1109/ICIEA.2010.5515585.

[13] MYAING, Aung; DINAVAHI, Venkata. FPGA-based real-time emulation of power electronic systems with detailed representation of device characteristics. In: *2011 IEEE Power and Energy Society General Meeting*. 2011, s. 1–11. Dostupné z DOI: 10.1109/PES.2011.6039125.

[14] PETRIC, Ivan Z.; MATTAVELLI, Paolo; BUSO, Simone. Multi-Sampled Grid-Connected VSCs: A Path Toward Inherent Admittance Passivity. *IEEE Transactions on Power Electronics*. 2022, roč. 37, č. 7, s. 7675–7687. Dostupné z DOI: 10.1109/TPEL.2022.3145191.

[15] NAOUAR, Mohamed-Wissem; MONMASSON, Eric; NAASSANI, Ahmad Ammar; SLAMA-BELKHODJA, Ilhem; PATIN, Nicolas. FPGA-Based Current Controllers for AC Machine Drives—A Review. *IEEE Transactions on Industrial Electronics*. 2007, roč. 54, č. 4, s. 1907–1925. Dostupné z DOI: 10.1109/TIE.2007.898302.

[16] *Vivado Design Suite User Guide: Design Flows Overview*. Online. Advanced Micro Devices, Inc. UG892 (v2022.1) April 20, 2022. Dostupné z: https://docs.xilinx.com/r/2022.1-English/ug892-vivado-design-flows-overview. [cit. 2023-12-30].

[17] *Vivado Design Suite User Guide: Using Constraints*. Online. Advanced Micro Devices, Inc. UG903 (v2022.1) June 1, 2022. Dostupné z: https://docs.xilinx.com/r/2022.1-English/ug903-vivado-using-constraints. [cit. 2024-01-01].

[18] *Vivado Design Suite User Guide: Synthesis*. Online. Advanced Micro Devices, Inc. UG901 (v2022.1) June 6, 2022. Dostupné z: https://docs.xilinx.com/r/2022.1-English/ug901-vivado-synthesis. [cit. 2024-01-01].

[19] *Vivado Design Suite User Guide: Implementation*. Online. Advanced Micro Devices, Inc. UG904 (v2022.1) May 24, 2022. Dostupné z: https://docs.xilinx.com/r/2022.1-English/ug904-vivado-implementation. [cit. 2024-01-01].

[20] *Fixed-Point Designer: Model and optimize fixed-point and floating-point algorithms*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/fixedpoint/index.html. [cit. 2024-01-03].

[21] *Meet Timing Requirements Using Enable-Based Multicycle Path Constraints*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/hdlcoder/ug/enable-based-multicycle-constraints.html. [cit. 2024-01-02].

[22] *HDL Coder: Generate VHDL and Verilog code for FPGA and ASIC designs*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/hdlcoder/index.html. [cit. 2024-01-03].

[23] *Clarke Transform: Implement ab to aβ transformation*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/mcb/ref/clarketransform.html. [cit. 2024-01-06].

[24] BURIÁNEK, Tomáš. *Vektorové řízení PMSM motoru Yaskawa SGMM-A1S312 na platformě LaunchPad F28069M v prostředí Simulink/Matlab*. Online, bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta strojní, 2021. Dostupné z: http://hdl.handle.net/10467/97328. [cit. 2024-01-06].

[25] *Park Transform: Implement aβ to dq transformation*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/mcb/ref/parktransform.html. [cit. 2024-01-06].

[26] *Field-Weakening Control: Develop field-weakening control for a permanent magnet synchronous motor (PMSM)*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/solutions/electrification/field-weakening-control.html. [cit. 2024-01-07].

[27] *Inverse Park Transform: Implement dq to aβ transformation*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/mcb/ref/inverseparktransform.html. [cit. 2024-01-06].

[28] *PWM Reference Generator: Generate modulated signals from phase voltages*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/mcb/ref/pwmreferencegenerator.html. [cit. 2024-01-07].

[29] *Product - FPGA: PYNQ™-Z2*. Online. TUL Corporation. Dostupné z: https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html. [cit. 2024-01-04].

[30] *PYNQ: PYTHON PRODUCTIVITY*. Online. Advanced Micro Devices, Inc. Dostupné z: http://www.pynq.io/. [cit. 2024-01-04].

[31] *PYNQ Introduction*. Online. Advanced Micro Devices, Inc., 2022. Revision a056b845. Dostupné z: https://pynq.readthedocs.io/en/v3.0.0/. [cit. 2024-01-04].

[32] *Development Boards: Downloadable PYNQ images*. Online. Advanced Micro Devices, Inc. Dostupné z: http://www.pynq.io/board.html. [cit. 2024-01-04].

[33] *Jupyter: Free software, open standards, and web services for interactive computing across all programming languages*. Online. Dostupné z: https://jupyter.org/. [cit. 2024-01-04].

[34] *BOOSTXL-DRV8305EVM: DRV8305N 3-Phase Motor Drive BoosterPack Evaluation Module*. Online. Texas Instruments Incorporated. Dostupné z: https://www.ti.com/tool/BOOSTXL-DRV8305EVM. [cit. 2024-01-04].

[35] *BOOSTXL-DRV8305EVM User's Guide*. Online. Texas Instruments Incorporated. SLVUAI8A–August 2015–Revised June 2017. Dostupné z: https://www.ti.com/lit/pdf/slvuai8. [cit. 2024-01-04].

[36] *DRV8305 Three Phase Gate Driver With Current Shunt Amplifiers and Voltage Regulator*. Online. Texas Instruments Incorporated. SLVSCX2B–AUGUST 2015–REVISED FEBRUARY 2016. Dostupné z: https://www.ti.com/lit/gpn/drv8305. [cit. 2024-01-04].

[37] *Industrial-Grade NEMA 23 Motors: Brushless, Permanent Magnet, Rotary Motors*. Online. Teknic, Inc. VERSION 6.3. Dostupné z: https://www.teknic.com/files/product_info/N23_Industrial_Grade_Motors_v6.3.pdf. [cit. 2024-01-04].

[38] *HUDSON BRUSHLESS DC SERVO MOTORS: TECHNICAL BROCHURE*. Online. Teknic, Inc. VERSION 2.7 MARCH 13, 2020. Dostupné z: https://www.teknic.com/files/downloads/Hudson%20User%20Manual.pdf. [cit. 2024-01-04].

[39] *Clocking Wizard*. Online. Advanced Micro Devices, Inc. Version 6.0 (Rev. 10). Dostupné z: https://www.xilinx.com/products/intellectual-property/clocking_wizard.html. [cit. 2024-01-10].

[40] *PYNQ-Z2: Schematic*. Online. TUL Corporation. Rev R10, Tuesday, January 09, 2018. Dostupné z: https://dpoauwgwqsy2x.cloudfront.net/Download/TUL_PYNQ_Schematic_R12.pdf. [cit. 2024-01-10].

[41] *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter: User Guide*. Online. Advanced Micro Devices, Inc. UG480 (v1.11) June 13, 2022. Dostupné z: https://docs.xilinx.com/r/en-US/ug480_7Series_XADC. [cit. 2024-01-10].

[42] *XADC Wizard*. Online. Advanced Micro Devices, Inc. Version 3.3 (Rev. 8). Dostupné z: https://www.xilinx.com/products/intellectual-property/xadc-wizard.html. [cit. 2024-01-10].

[43] *Generate FPGA Block RAM from Lookup Tables*. Online. The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/hdlcoder/ug/generate-fpga-block-ram-lookup-tables.html. [cit. 2024-01-12].

[44] *Zynq 7000 Processing System IP*. Online. Advanced Micro Devices, Inc. Version 5.5 (Rev. 6). Dostupné z: https://www.xilinx.com/products/intellectual-property/processing_system7.html. [cit. 2024-01-13].

[45] *AXI General Purpose IO*. Online. Advanced Micro Devices, Inc. Version 2.0 (Rev. 28). Dostupné z: https://www.xilinx.com/products/intellectual-property/axi_gpio.html. [cit. 2024-01-13].

[46] *AXI Direct Memory Access*. Online. Advanced Micro Devices, Inc. Version 7.1 (Rev. 27). Dostupné z: https://www.xilinx.com/products/intellectual-property/axi_dma.html. [cit. 2024-01-13].

[47] *AMBA AXI4 Interface Protocol: AXI Details*. Online. Advanced Micro Devices, Inc. Dostupné z: https://www.xilinx.com/products/intellectual-property/axi.html#details. [cit. 2024-01-14].

[48] *AXI DMA: LogiCORE IP Product Guide*. Online. Advanced Micro Devices, Inc. PG021 April 27, 2022. Dostupné z: https://docs.xilinx.com/r/en-US/pg021_axi_dma. [cit. 2024-01-13].

[49] *AMBA® 4 AXI4-Stream Protocol: Specification*. Online. ARM, 2010. Version: 1.0. Dostupné z: https://developer.arm.com/documentation/ihi0051/a/. [cit. 2024-01-14]. ARM IHI 0051A (ID030610).

[50] *DMA*. Online. Advanced Micro Devices, Inc., 2022. Revision a056b845. Dostupné z: https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries/dma.html. [cit. 2024-01-15].

[51] *Flet: The fastest way to build Flutter apps in Python*. Online. Dostupné z: https://flet.dev/. [cit. 2024-01-15].

[52] *asyncio: Asynchronous I/O*. Online. Dostupné z: https://docs.python.org/3.10/library/asyncio.html. [cit. 2024-01-15].

[53] *Downloads: Vivado ML Edition*. Online. Advanced Micro Devices, Inc. Dostupné z: https://www.xilinx.com/support/download.html. [cit. 2024-01-18].

[54] *Board Settings: Vivado board files*. Online. Advanced Micro Devices, Inc., 2022. Revision a056b845. Dostupné z: https://pynq.readthedocs.io/en/v3.0.0/overlay_design_methodology/board_settings.html#vivado-board-files. [cit. 2024-01-18].

[55] *Writing an SD Card Image*. Online. Advanced Micro Devices, Inc., 2022. Revision a056b845. Dostupné z: https://pynq.readthedocs.io/en/v3.0.0/appendix/sdcard.html#writing-the-sd-card. [cit. 2024-01-18].

[56] *PYNQ-Z2 Setup Guide*. Online. Advanced Micro Devices, Inc., 2022. Revision a056b845. Dostupné z: https://pynq.readthedocs.io/en/v3.0.0/getting_started/pynq_z2_setup.html. [cit. 2024-01-18].

[57] *Python 3.10.8*. Online. Python Software Foundation. Release Date: Oct. 11, 2022. Dostupné z: https://www.python.org/downloads/release/python-3108/. [cit. 2024-01-18].

[58] *Poetry*. Online. Version 1.4.2. Dostupné z: https://python-poetry.org/. [cit. 2024-01-18].

C Seznam obrázků

D Seznam tabulek