České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

BAKALÁŘSKÁ PRÁCE

Využití programovatelného pole pro řízení bezkartáčových motorů

Vypracoval: Vladimír Burian
Vedoucí práce: Ing. Pavel Píša, Ph.D.

Praha, 2011

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Vladimír Burian**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **Využití programovatelného pole pro řízení bezkartáčových motorů**

Pokyny pro vypracování:

1. Seznamte se s architekturami programovatelných obvodů XILINX a nástroji pro logický návrh v jazyce VHDL.
2. Zvolte vhodný obvod pro realizaci polohového regulátoru třífázového Brushless DC motoru vybaveného inkrementálním čidlem, který by obsahoval alespoň 24-bitový čítač polohy, generátor požadované polohy, PSD regulátor a třífázový PWM generátor.
3. Dále proveďte logický návrh pro vybraný obvod, který bude zahrnovat navržené periferie pro řízení motoru a syntetizovaný mikrokontrolér. Mikrokontrolér bude zajišťovat komunikaci s počítačem třídy PC a vyšší úrovně řízení.

Seznam odborné literatury:

[1] Materiály k předmětům X35MSY a A0B35SPS Katedry řízení, ČVUT FEL
[2] Volnei A. Pedroni: Digital Electronics and Design with VHDL, MORGAN KAUFMANN 2008, ISBN: 0123742706
[3] Enoch O. Hwang: Digital Logic and Microprocessor Design with VHDL, Thomson 2006, ISBN: 0-534-46593-5

Vedoucí: Ing. Pavel Píša, Ph.D.

Platnost zadání: do konce zimního semestru 2011/2012

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 14. 1. 2011

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 24.05.2011 .....

podpis

Poděkování

Na tomto místě bych chtěl především velmi poděkovat mému vedoucímu bakalářské práce panu Ing. Pavlu Píšovi, Ph.D. za ochotu, cenné rady, věcné připomínky a trpělivost v průběhu jejího řešení.

Abstrakt

Bakalářská práce se zabývá řízením bezkartáčových motorů pomocí programovatelných hradlových polí. Nejvyšší úroveň řízení zajišťuje syntetizovaný mikrokontrolér především s knihovnou PXMC pro řízení motorů, jež byla vyvinuta ve firmě PiKRON. Komunikace s nadřazeným systémem probíhá po standardní sériové lince RS232 v jednoduchém textovém protokolu. Na nejnižší úrovni samočinně provádí komutaci k tomu navržený syntetizovaný hardware, jež je řízen nadřazeným mikrokontrolérem. Implementována je komutace modifikovaným sinusovým signálem, ale návrh svou koncepcí umožňuje doplnění proudovými regulátory a momentovým řízením. Kromě samotné implementace je v práci také obecně rozebrán princip funkce bezkartáčového motoru a možnosti jeho řízení. Následuje popis architektury hradlových polí firmy Xilinx, jež byli při realizaci použity. Nechybí ani část věnovaná popisu vývojových nástrojů, postupu vývoje a jazykům k němu určeným.

Abstract

The bachelor thesis presents control of brush-less DC motors with use of field programmable gate arrays – FPGA. The highest level of control is handled by softcore MCU running PXMC motion control library developed in PiKRON. MCU communicates with master system through a standard serial line RS232 using simple text-based and human-readable protocol. At the bottom level there is synthesized hardware doing motor commutation and being controlled by MCU. Method of modified sine wave commutation is implemented, but the design is proposed to be simply extensible by current controllers and so torque control. Beside of implementation there is described general operation of BLDC motors and basic principles of their control. Description of Xiling FPGA, which was used to implement design, is following. Few sections are also intended for development tools, synthesis process and hardware description languages.

Obsah

Kapitola 1

Úvod

Elektrické pohony jsou nezbytnou součásti mnoha strojů, robotů a dalších zařízení. Stejně jako ostatní komponenty i motory jsou často pečlivě vybírány podle druhu cílové aplikace, prostředí a povaze provozu a dalších kritérií. Mezi možnými alternativami zaujímají bezkartáčové motory pevné místo pro mnohé své výjimečné vlastnosti. Avšak množství výhod, které nám přináší, je vyváženo některými nevýhodami. Stejnosměrné bezkartáčové motory spadají do skupiny tzv. elektronicky komutovaných motorů. A právě nutnost a relativní složitost zajištění komutace elektronicky je často zmiňovanou nevýhodou. Nicméně i tato nevýhoda nemusí být ve výsledku nevýhodou protože nám naopak dovoluje implementovat řízení dokonalejší tam, kde je ho potřeba, a dále tím vlastnosti pohonu oproti ostatním možnostem vylepšit.

Implementace elektronické komutace je pro bezkartáčové motory klíčová. Pokud má motor sloužit jako servopohon, pak vyžaduje použití výpočetní techniky a periodické a velmi časté vykonávání stejné posloupnosti akcí. Dnes je jejich řízení nejčastěji realizováno v jednočipových mikropočítačích, které mají integrovány periferie poskytující PWM výstupy a v některých případech i vstupy pro vyhodnocení signálů inkrementálních rotačních kodérů. Tedy vše potřebné pro rozumné polohové řízení běžných bezkartáčových motorů. Bohužel počet dostupných integrovaných periférií je většinou postačující k obsluze pouze jediného motoru. Možností, jak rozšířit počet řízených motorů jedním mikropočítačem, je např. připojení dodatečných externích periferií. Jako jiné, zajímavé řešení se jeví implementace těchto periférií v programovatelném hradlovém poli, přičemž by hradlové pole mohlo převzít celou část stále se opakujícího procesu komutace i pro několik motorů a zmenšit tak zatížení mikropočítače, který by se poté mohl intenzivněji věnovat řízení na vyšších úrovních, plánování pohybu, komunikaci s nadřazenými systémy apod. Právě touto alternativou se zabývá následující práce.

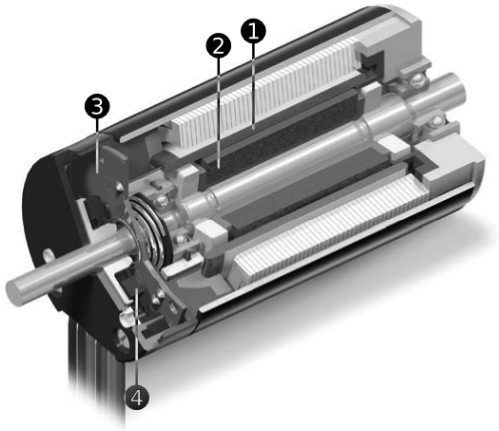
V první kapitole je rozebrána funkce bezkartáčových motorů a běžné způsoby jejich řízení. V kapitole druhé jsou pak rozebrány funkce a možnosti hradlových polí firmy Xilinx. Taktéž jsou představeny vývojové nástroje, postup syntetizace hardwaru, jazyky pro jeho popis a několik rad, které je dobré při návrhu znát. Poslední dvě kapitoly se věnují implementaci hardwaru a softwaru. Popis implementace se v žádném případě nesnaží být vyčerpávající, ale měl by být dostatečný k pochopení fungovaní celého projektu. Důraz je kladen spíše na vysvětlení celkové koncepce, nastínění možností, které se při implementaci nabízí, jejich porovnání a zdůvodnění volby.

Kapitola 2

Bezkartáčové motory

Na obrázku 2.1 můžeme vidět řez stejnosměrným bezkartáčovým motorem z katalogu firmy Maxon [9]. Minimálním, obecným základem, který charakterizuje tuto třídu motorů, je stator (tělo motoru) nesoucí statorové vinutí, jež je přímo řízeno řídicí jednotkou, a rotor s permanentním magnetem. Interakcí mezi vinutím a permanentním magnetem vzniká točivý moment. (Nutno podotknout, že ne vždy je vnější částí motoru stator. Existují i konstrukce, kde je vnější část tvořena rotorem. Tyto se vyznačují větším krouticím momentem a větším momentem setrvačnosti. Jsou tedy vhodnější do aplikací, kde se vyžaduje spíše stabilní rychlost otáčení, případně kde je vhodný jejich charakteristický diskový tvar.)

Jak již bylo řečeno v úvodu, podstatou funkce je, že procesy ve statorovém vinutí má



Obrázek 2.1: Řez bezkartáčovým motorem firmy. MAXON.
1 - samonosné vinutí statoru, 2 - permanentní magnet rotoru, 3 - deska s Hallovými senzory, 4 - permanentní magnety na rotoru spínající Hallovy senzory

plně pod kontrolou řídicí jednotka, která se musí starat o tzv. elektronickou komutaci – řízení napětí na vývodech vinutí za účelem vyvození točivého momentu. Od tohoto faktu se odvíjí většina výhod i nevýhod bezkartáčových motorů. Komutaci lze prováděl různými způsoby, některé z nich rozebereme v následující části. Důležité je, že za tímto účelem potřebujeme měřit natočení rotoru. Proto bývají motory vybaveny senzory.

Jako základní senzory jsou nejčastěji použity Hallovy sondy integrované přímo v těle motoru. Ty snímají buď magnetické pole hlavního magnetu rotoru, nebo magnetické pole pomocného magnetu, jak můžeme vidět v uvedeném řezu. Počet těchto senzorů je shodný s počtem fází. Jejich geometrické uspořádání může být různé, vždy nám ale poskytují informaci o elektrické fázi rotoru v mezích rovných čtvrtině rozestupu vinutí (za ideálního případu). Pro 3-fázový motor s dvoupólovým rotorem, který má fázové rozestupy statorových vinutí 120°, měří Hallovy sondy fázi rotoru s přesností ±30°.

Pokud nejsou Hallovy sondy pro danou aplikaci dostačující, přidává se k motoru senzor s větším rozlišením. Velmi často je to optický inkrementální senzor, který běžně poskytuje rozlišení $0,2°$. Poskytuje nám ale pouze informaci o relativní změně polohy, nikoli přímo absolutní polohu jako Hallovy senzory. Nemůže je tedy zcela nahradit, zpravidla proto oba dva vystupují v součinnosti. Kromě těchto senzorů se můžeme v této funkci setkat i s méně častými senzory, jako jsou magnetické inkrementální senzory či dokonce resolvery.

Následuje výčet pozitivních vlastností těchto motorů plynoucí z jejich konstrukce a možného způsobu řízení:

- Prostorově úsporné, velký poměr výkon/velikost.
- Spolehlivé, bezúdržbové – neobsahují mechanický komutátor, který podléhá opotřebení a je nejslabší částí klasických stejnosměrných motorů. Tím odpadá i jiskření a uvolňování částic z komutátoru, což může být v některých případech nežádoucí.
- Vinutí je spojeno s tělem motoru a dobře se chladí, i když je motor zcela uzavřený.
- Řízením lze dosáhnout malého kolísání kroutícího momentu v průběhu jedné otáčky. To má za následek i malou hlučnost motoru, omezení vyvolávaných vibrací a celkově kultivovanější chod.
- Krouticí moment je lineálně závislý na proudu. Motory MAXON mají např. samonosná vinutí bez ocelových koster. Minimalizují se tím nelinearity jinak způsobené pólovými nástavci, zároveň mají vinutí malou indukčnost.
- Velký rozběhový moment.
- Velká účinnost.

To jsou důvody, proč můžeme vidět bezkartáčové motory nejen na kritických místech

v průmyslu, letecké a kosmické technice, nemocničních aplikacích apod.

2.1 Rozbor funkce

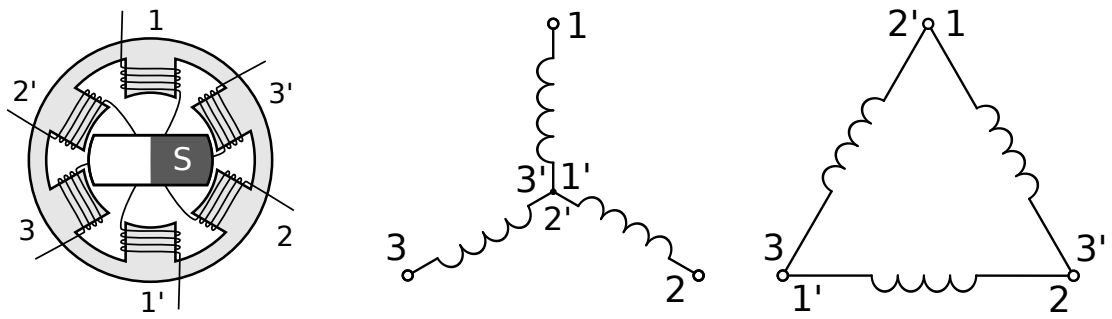
Funkci pro jednoduchost vysvětlíme na 3-fázovém bezkartáčovém motoru s 2-pólovým statorem. Schéma vnitřního uspořádání můžeme vidět na obrázku 2.2. Proud vinutími, respektive napětí na nich lze vyjádřit jako vektor složený z dílčích vektorů příslušných daným cívkám s nimiž sdílejí fyzickou orientaci. Na Obrázku 2.3 jsou tímto způsobem znázorněny možné poměry v motoru za ustáleného stavu odpovídacího náčrtku 2.2.

Vznik točivého momentu lze popsat na základě Ampérova zákonu magnetické síly jako sílu působící na vodič protékaný proudem v magnetickém poli:

$$M = k_M \cdot i \cdot \sin \varphi_i \tag{2.1}$$

Kde φ_i je fázový posuv mezi vektorem magnetická indukce $\mathbf{B}$ rotoru (který symbolizuje především jeho natočení) a vektorem proudu. Konstanta točivého momentu $k_M(\mathrm{NmA}^{-1})$ charakterizuje konstrukci motoru z elektromechanického hlediska jako točivý moment, jež lze vyvolat jednotkou proudu. (Okamžitý točivý moment v průběhu jedné otáčky nemusí vždy přesně odpovídat uvedenému vztahu. Příčinou je deformace mg. pole rotoru pólovými nástavci statoru. Někteří výrobci se ale snaží ideálního stavu dosáhnout a daří se jim to např. použitím již zmíněného samonosného vinutí, které žádné pólové nástavce nemá.) Pro nás je v tuto chvíli důležité, že největšího točivého momentu za daného proudu dosáhneme, pokud je vektor proudu otočený o 90° proti mg. poli rotoru.

Jak závisí proud na budicím napětí můžeme vypočítat ze zjednodušeného modelu



(a) Prostorové uspořádání (b) Zapojení vinutí do hvězdy a trojúhelníku

Obrázek 2.2: Schématické znázornění konstrukce 3-fázového bezkartáčového motoru s dvoupólovým rotorem.

motoru tvořeného ideálním odporem a indukčností spolu s ideálním motorem, který představuje zpětně indukované napětí.

$$\mathbf{u} + \mathbf{u_i} = R\mathbf{i} + L\mathbf{i}' \tag{2.2}$$

Napětí indukované v motoru $\mathbf{u_i}$ odpovídá derivaci magnetického toku vinutím $\mathbf{B}$. Jestliže orientace vektoru magnetické indukce odpovídá vektoru $e^{j\omega t}$, tak můžeme psát:

$$\mathbf{u_i} = -\frac{d\phi}{dt} = -k_M(e^{j\omega t})' = -j\omega k_M e^{j\omega t} \tag{2.3}$$

Dosazením do napěťové rovnice (2.2) a zapsáním vektorů v exponenciálním tvaru vychází:

$$ue^{j(\omega t+\varphi_u)} - j\omega k_M e^{j\omega t} = Rie^{j(\omega t+\varphi_i)} + L(ie^{j(\omega t+\varphi_i)})' \tag{2.4a}$$

$$\mathbf{u_r}e^{j\omega t} - j\omega k_M e^{j\omega t} = R\mathbf{i_r}e^{j\omega t} + L(\mathbf{i_r}e^{j\omega t})' \tag{2.4b}$$

$$\mathbf{u_r}e^{j\omega t} - j\omega k_M e^{j\omega t} = R\mathbf{i_r}e^{j\omega t} + j\omega L\mathbf{i_r}e^{j\omega t} + L\mathbf{i_r}'e^{j\omega t} \tag{2.4c}$$

$$\mathbf{u_r} - j\omega k_M = R\mathbf{i_r} + j\omega L\mathbf{i_r} + L\mathbf{i_r}' \tag{2.4d}$$

V poslední rovnici (2.4d) již je vztah, který popisuje dynamickou závislost proudu, napětí a otáček, přičemž napětí a proud jsou vyjádřeny jako vektory $\mathbf{u_r}$, $\mathbf{i_r}$ relativní k vektoru $e^{j\omega t}$, který má shodnou orientaci jako $\mathbf{B}$. Ustálený stav je potom popsán rovnicí (2.5) jejíž grafickou interpretaci můžeme vidět na diagramu 2.3.

$$\mathbf{u_r} - j\omega k_M - R\mathbf{i_r} - j\omega L\mathbf{i_r} = 0 \tag{2.5}$$

Vidíme, že v obecném případě nemá vektor proudu s vektorem napětí stejnou orientaci, což je způsobeno indukčností motoru. Pokud bychom stator budili napětím kolmým k $\mathbf{B}$, jak ukazuje diagram 2.3(a), tak pro fázový posuv proudu platí:

$$\varphi_i = \arctan\frac{R}{\omega L} \tag{2.6}$$

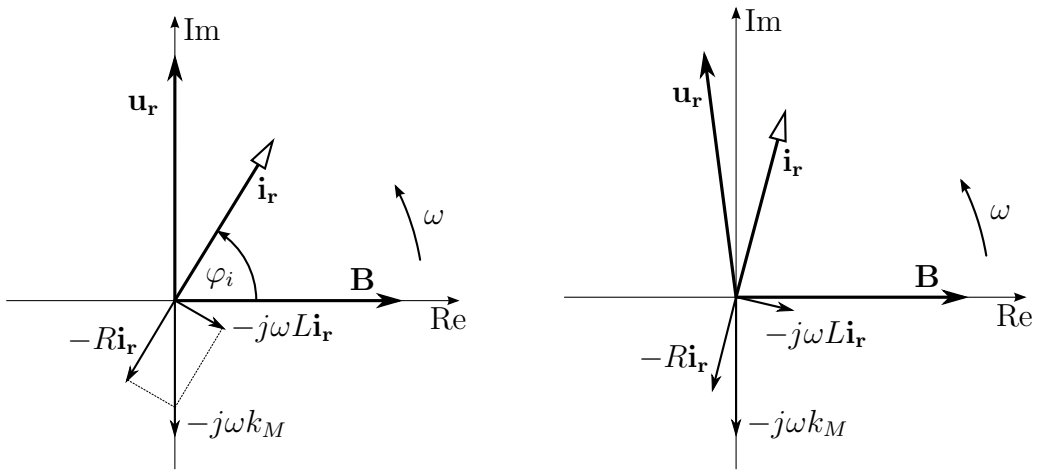
Zkusme se podívat na konkrétní čísla pro motor z nabídky Maxon s nominálními parametry: 40 W, 12 V, 5,37 A a 27000 ot.min^{-1}, dvoupólový rotor. Katalog dále prozrazuje, že $L = 0,0158\,\mathrm{mH}$, $R = 0,204\,\Omega$, $k_M = 0,00375\,\mathrm{NmA^{-1}}$ a $J = 239 \cdot 10^{-9}\,\mathrm{kgm^2}$. Pro případ buzení kolmým vektorem napětím bude za nominálních otáček fázový posuv proudu následující:

$$\varphi_i = \arctan \frac{0,204}{27000/60 \cdot 2\pi \cdot 0,0158 \cdot 10^{-3}} = 77,6^\circ \tag{2.7}$$

To znamená, že vektor proudu je oproti ideálnímu případu posunut o -12.4°. Což můžeme interpretovat tak, že se krouticí moment sníží na 97.7% ($\sin 77.6^\circ = 0,977$). To ale není nikterak dramatické číslo.

Důležité je si uvědomit, že největší úbytek napětí v obvodu tvoří indukované napětí. (A to je dobře, protože to poukazuje na dobré vlastnosti motoru.) V našem případě tvoří celých $k_M\omega = 10,6\,\mathrm{V}$. (Z tohoto hlediska je diagram 2.3 poněkud zavádějící.) Neméně důležité je, že samozřejmě působí proti budicímu napětí, ale přitom nemění svou orientaci, stále je kolmé k $\mathbf{B}$. Problém tedy může nastat, pokud budicí napětí kolmé není. V takovém případě je jeho reálná složka $\mathbf{u_r}$ (podle diagramu 2.3) kompenzována pouze napěťovým úbytkem na RL prvcích a v důsledku může velmi dramaticky zvýšit nechtěnou reálnou složku protékajícího proud, snížit účinnost atd. Tento jev můžeme pro náš příkladový motor vidět na grafech 2.4.

Vhodným zvýšením fázového posuvu budicího napětí (zvětšením předstihu) můžeme samozřejmě naopak úbytek na RL prvcích minimalizovat a dosáhnout kolmého vektoru proudu. (I u některých kartáčový motorů je možné mechanicky seřídit předstih komutátoru na optimální výsledek.) Ale to si vyžaduje poměrně přesná měření v konkrétní aplikaci. Bez nich je jistější udržovat kolmý pouze vektor napětí. V ukázkovém případu by např. stačil dodatečný posun o pouhých $0,6^\circ$ (viz. grafy 2.4).



(a) Buzení proudem kolmým k vektoru mg. indukce

(b) Buzení obecným proudem

Obrázek 2.3: Fázové diagramy proudů a napětí v motoru za ustáleného stavu.

2.2 Metody řízení

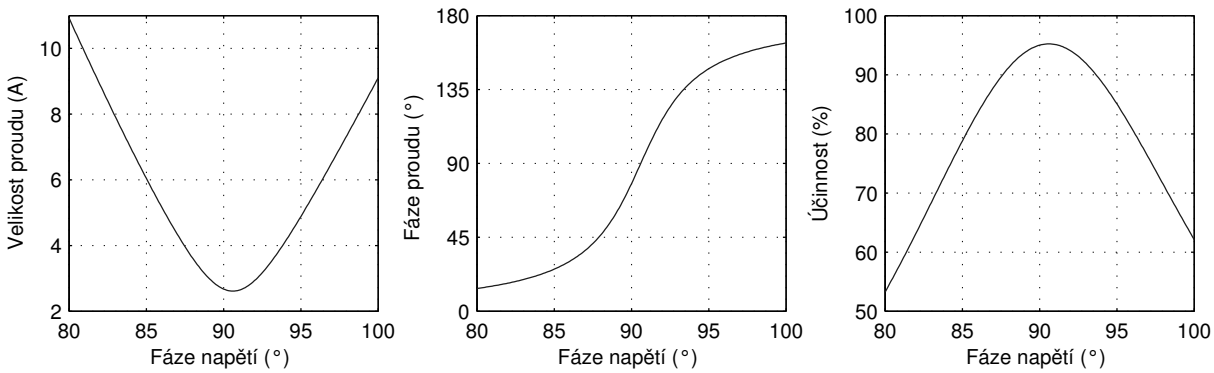
V následujících odstavcích jsou uvedeny běžné způsoby řízení komutace bezkartáčových motorů.

2.2.1 Bloková komutace

Bloková komutace je pravděpodobně nejjednodušší možná realizace elektronické komutace vůbec. Nedělá nic jiného, než že zcela kopíruje funkci klasického komutátoru. Jedná se tak de facto o emulaci obyčejného stejnosměrného motoru. K snímání polohy rotoru slouží v tomto režimu Hallovy senzory. Jak mohou vypadat průběhy vstupů a výstupů vidíme na obrázku 2.5, který koresponduje s obrázkem 2.2.

Jeden komutační cyklus je rozdělen na 6 bloků. V každém bloku je vždy jedna fáze sepnuta proti zemi, jedna fáze je sepnuta proti napájecímu napětí a zbývající fáze není sepnuta proti ničemu – její případný proud teče přes ochranné diody a klesá k nule, stejně jako když uhlíky mechanického komutátoru přemostí jeho sousední lamely. Jedna z aktivních fází (myšleno fáze sepnutá proti zemi nebo napájecímu napětí) je v daném komutačním bloku navíc místo trvalého sepnutí obvykle buzena PWM signálem, čímž je motor regulován.

Mezi přední výhody toho řešení patří jeho jednoduchost, řídicí obvod není nutné nikterak inicializovat, po zapnutí může okamžitě plnohodnotně pracovat, a to v celém rozsahu možných otáček. Nevýhodou je nedokonalost komutace. Protože je komutačních bloků 6, je napětí udržováno v předstihu skokově v rozmezí $90 \pm 30°$ (za ideálního stavu). Navíc poloha Hallových senzorů samozřejmě není naprosto přesná a jejich logika pracuje s hysterezí kvůli potlačení šumu, jejich výstup je tedy mírně opožděný. Proto ve výsledku



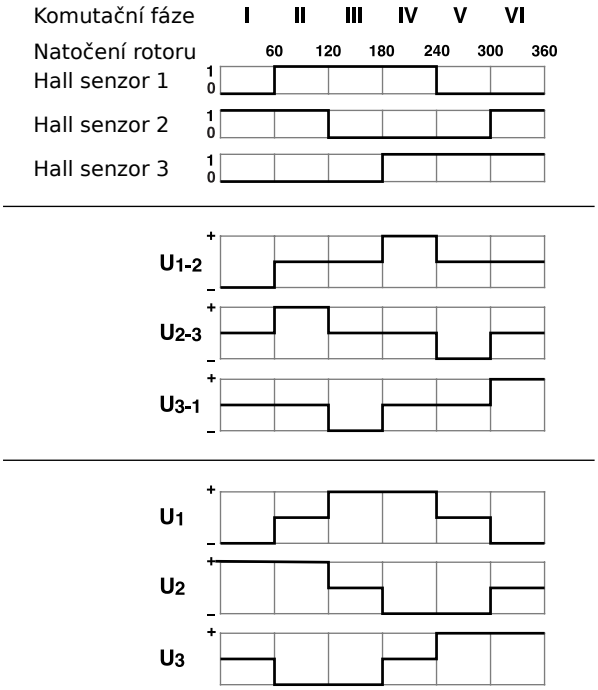
Obrázek 2.4: Závislosti proudu a účinnosti (bez započítání mechanických ztrát) na fázovém posuvu budicího napětí při plných otáčkách a polovičním točivém momentu vzhledem k nominálním hodnotám.

klesá účinnost pohonu, točivý moment je zvlněný (udává se 14 %), vznikají vibrace atd.

2.2.2 Bezsenzorová komutace

V principu se také jedná o komutaci po blocích jako v předchozím případě, ale liší se ve způsobu určení polohy rotoru. Bezsenzorovostí je v tomto významu myšleno, že motor sám o sobě žádný senzor neobsahuje. Avšak řídicí jednotka musí měřit napětí alespoň na jednom vinutí. V komutačním bloku, kde toto vinutí není sepnuto, se sleduje okamžik, kdy v něm indukované napětí projde nulou. Tato událost značí, že rotor byl právě orientován shodně s tímto vinutím.

Řízení tímto způsobem je komplikovanější. Motor musím mít určité minimální otáčky, aby bylo možné spolehlivě měřit průchody indukovaného napětí nulou a tím pádem i spolehlivě komutovat. Motor tedy za nízkých otáček stěží pracuje a jeho rozběh je problémový. Rozhodně tedy není vhodný do polohovacích a jiných hodnotě dynamických aplikací. Uplatnění najde především tam, kde motor z nějakého důvodu Hallovy senzory nemá, nebo je mít ani nemůže.



Obrázek 2.5: Ukázka průběhů signálů z Hallových senzorů a symbolické znázornění napětí na cívkách a jednotlivých fázích. Úroveň mezi + a − značí rozepnutí spících prvků.

2.2.3 Komutace sinusovým signálem

Při tomto způsobu komutace je napětí rozloženo na všechny fáze tak, že velikost celkového vektoru napětí na motoru se nemění (pokud to sami nepožadujeme) a předstih je konstantní, zpravidla 90° plus případná korekce indukčnosti. Budicí napětí v ustáleném stavu mají tvar sinusovek se shodnými amplitudami a stejnými fázovými rozestupy – stejně jako např. 3-fázové napětí v rozvodné síti. Takto můžeme teoreticky optimálně využít možnosti motoru.

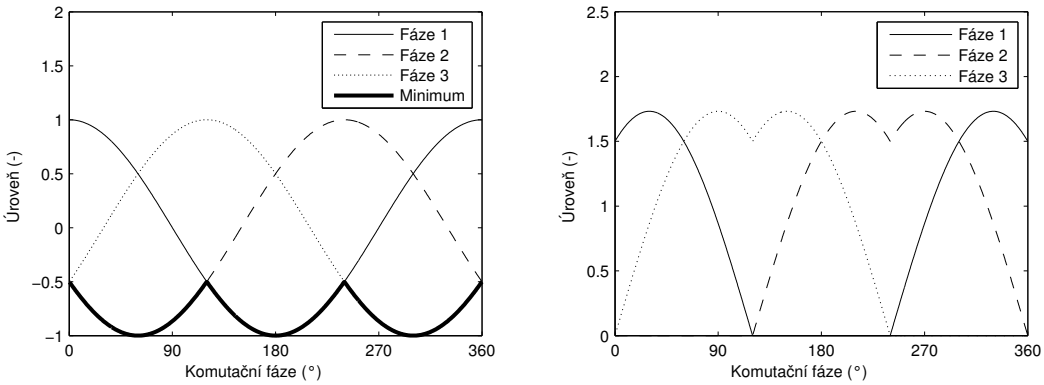
Nutností je v tomto případě samozřejmě poněkud přesnější měření polohy rotoru než v případě blokové komutace. V praxi je fáze rotoru kvantována např. na 1000 hodnot za komutační otáčku, což je poměrně dostačující. Při použití např. 10-bitových PWM generátorů coby budičů jsou budicí napětí kvantována na 1024 úrovní – také dostačující. Problém by mohl nastat s časovým kvantováním. Běžná je frekvence PWM signálu okolo 20 kHz. Pokud se vrátíme k našemu ukázkovému motoru, tak za nominálních otáček 27000 ot.min^{-1} vychází necelých 49 komutačních cyklů na jednu komutační periodu motoru, což už není tolik, ale pro funkci stále stačí.

Situace ve skutečnosti tedy není úplně ideální, ale velmi dobře se jí blížíme. Tímto způsobem využijeme většinu možností, které bezkomutátorové motory nabízejí, jako velký rozběhový moment, hladký chod, velkou účinnost. Hlavní nevýhodou je složitější (a dražší) řídicí elektronika a nutnost přesného snímání polohy rotoru. Běžně používaný IRC senzor je navíc relativní senzor, takže na začátku je potřeba ho správně zarovnat např. podle známé polohy indexní značky. Do té doby se využívá bloková komutace. Méně časté řešení je použití absolutního senzoru (např. optického, nebo resolveru), které tímto problémem netrpí. Tím se ale dostáváme do vyšších a značně specializovaných kategorií produktů.

2.2.4 Modifikovaná sinusovka

Jedná se o způsob realizace napěťového buzení motoru, konkrétně buzení sinusovými průběhy, ale uvedený postup je použitelný obecně. Pointa je v tom, že požadované napětí na jednotlivých fázích je nutno chápat jako napětí oproti středu vinutí, tj. jako napětí na daných cívkách. Střed vinutí přitom nemusí mít nulový, ani konstantní potenciál. Proto je možné fázi (příp. fáze) s minimálním požadovaným napětím sepnout proti zemi úplně a zbylé fáze budit o to menším napětím – všechny napětí patřičně posunout. Situaci znázorňují grafy 2.6. Přestože na vinutích jsou stále sinusová napětí, tak napětí na fázích připomínají sinus jen velmi vzdáleně – mluvíme o tzv. modifikované sinusovce.

Počet sepnutí všech tranzistorů se tak o třetinu sníží a tím o necelou třetinu klesne



(a) Obvyklé 3-fázové napětí

(b) Modifikované sinusovky

Obrázek 2.6: Názorná ukázka vzniku tzv. modifikované sinusovky odečtením minima.

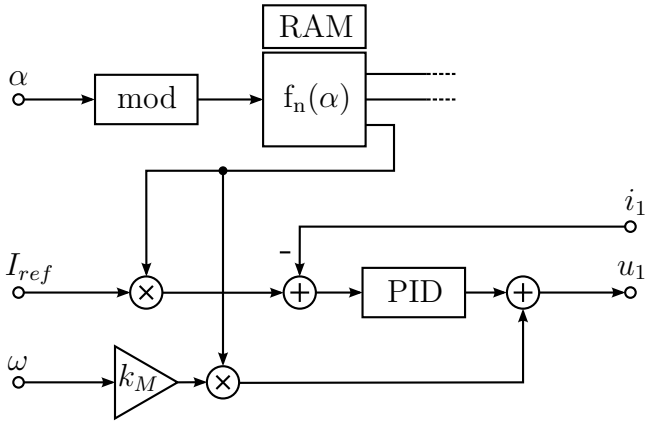
i ztrátový výkon na nich – k největším ztrátám u moderních, polem řízených tranzistorů dochází právě při změně jejich stavu. Zároveň můžeme dosáhnout o 16% větší amplitudy napětí na vinutích ($2/\sqrt{3} = 1.155$) bez zvýšení napájecího napětí. Tento princip je často využíván kdykoliv pracujeme s vícefázovým střídavým napětím, např. i ve frekvenčních měničích.

2.2.5 Momentové řízení

Při buzení napětím se vektor proudu nejenže opožďuje, ale navíc je jeho velikost velmi závislá na otáčkách, tedy i velikost točivého momentu se s otáčkami značně mění. A nejen s otáčkami, ale např. i s teplotou atd. V mnohých aplikacích to příliš nevadí. V robotice je ale možnost přímo řídit moment akčního členu velmi výhodná. Lze tak upotřebit znalost dynamiky robota a manipulovat s ním mnohem přesněji a citlivěji. Navíc je např. možné v jednom směru udržovat konstantní souřadnici a ve směru kolmém působit definovanou silou – nezbytnost nejen při strojovém obrábění. V praxi může být robot vybaven přímo senzorem kroutícího momentu a řídit tak kroutící moment zpětnovazebně přes něj, přesto se stále jedná o velmi přínosný a využívaný nástroj.

Možné řešení je naznačeno na blokovém diagramu 2.7. Postup řízení je následující:

Vstupem α je úhlová poloha motoru např. z kvadraturního čítače IRC senzoru. Následující blok s operací modulo naznačuje výpočet komutační fáze v intervalu 0 až 2π. Nemusí se nutně jednat o úplnou implementaci operace modulo, stačí udržovat offset vstupu oproti požadovanému výstupu a ošetřit případy přetečení a podtečení daného intervalu. Funkce $\mathrm{f}_n(\alpha)$ dále rozloží pomyslný jednotkový vektor s daným směrem do tří složek

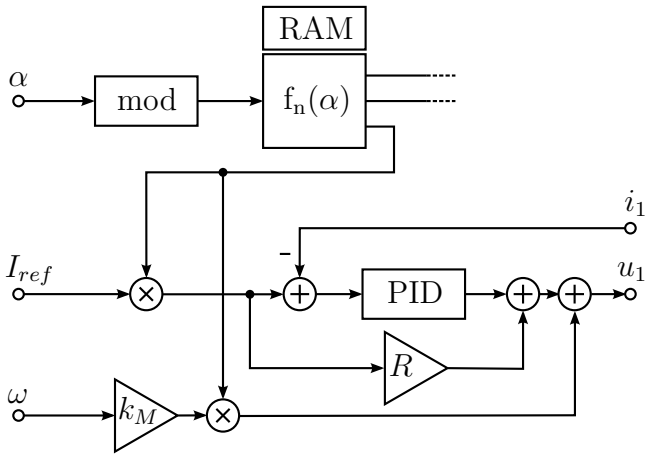


Obrázek 2.7: Blokový diagram možného momentového regulátoru.

odpovídajících třem fázím. K tomu je často využita předpočítaná tabulka v paměti. Vynásobením těchto složek referenční hodnotou amplitudy proudu I_{ref} (která přímo určuje točivý moment) získáme referenční hodnotu proudu pro každou z fází.

Následuje klasické zapojení PID regulátoru, které se snaží referenční hodnotu udržet. V ustáleném stavu tedy sleduje sinusový signál frekvencí odpovídající otáčkám. Číslicový PID regulátor má ale vzhledem k této skutečnosti poměrně malou šířku pásma, která se negativně projeví především při vyšších otáčkách. Do řetězce je proto zařazena přímá vazba z okamžitých otáček ω, která kompenzuje největší napěťový úbytek na motoru $-j\omega k_M$ (viz. diagram 2.3) a regulátoru tím pomáhá.

Další jednoduchá možnost, jak ještě více PID regulátoru usnadnit práci by bylo zařazení přímé vazby z referenčního proudu, která by částečně kompenzovala úbytek rovný $-R\mathbf{i_r}$, jak je znázorněno v blokovém diagramu 2.8.



Obrázek 2.8: Momentový regulátor doplněný o další přímou vazbu.

Na PID regulátor tak zbývá doregulovat nepřesnosti přímé vazby, vliv indukčnosti

motoru a další odchylky, se kterými není počítáno. A to už by pro něj neměl být problém. Tím spíš při menších otáčkách, kde se momentové řízení pravděpodobně více využije.

Nevýhody jsou zřejmé – řídicí systém je poměrně složitý, musí být relativně rychlý a musí nějakým způsobem měřit proud alespoň ve dvou fázích (proud třetí fází je předurčen). Výhody již byly jmenovány – v mnohých případech se jedná o nenahraditelný nástroj.

Kapitola 3

FPGA Xilinx

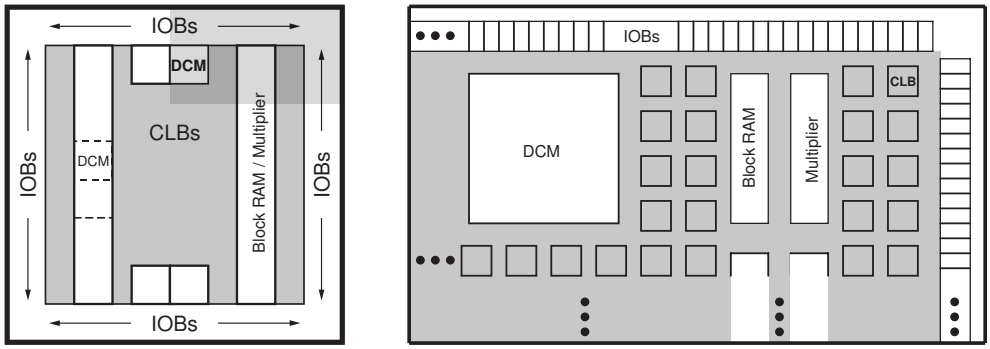
Programovatelná hradlová pole, označovaná jako FPGA (Field Programmable Gate Array), byla na trh uvedena firmou Xilinx v polovině 80. let. Jsou to obvody z třídy tzv. rekonfigurovatelné logiky (hardwaru) – v jednom pouzdře je mnoho různým logických bloků, jejichž funkce může být v omezené míře konfigurována stejně jako jejich vzájemné propojení. Primárním cílem architektury FPGA jsou aplikace vyžadující rozsáhlé a velmi rychlé logické obvody, které obvykle provádí výpočetně náročné algoritmy a s výhodou využívají paralelního zpracování dat. Mezi ně patří především oblast vysokorychlostních komunikací, síťových zařízení, šifrování, zpracování obrazů a jiných signálů, vědecké aplikace a další. Na trhu jsou dnes ale i cenově dostupné varianty obvodů určené pro méně náročné aplikace v automobilovém průmyslu či spotřební elektronice, jež vyhovují i našemu záměru. [8], [18]

3.1 Struktura architektury

V dalším textu je podrobněji popsaná architektura obvodů FPGA Xilinx. Obvod se skládají z propojovací sítě, logických (CLB) a vstupně výstupních bloků, případně dalších specializovaných prvků RAM, násobiček atd. Architektury FPGA obvodů jiných výrobců jsou, co se týče obecného rozdělení, podobné, ale použitá terminologie a granularita jednotlivých částí a komponent se většinou liší.

3.1.1 Konfigurovatelné logické bloky – CLB

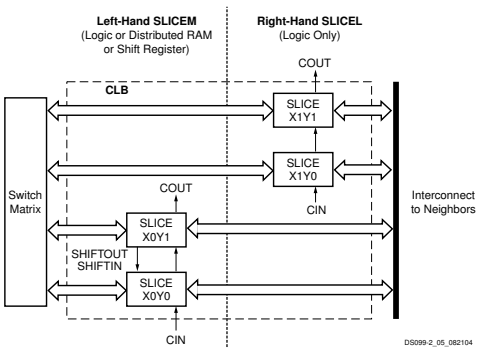
Největší část prostoru hradlového pole vyplňují konfigurovatelné logické bloky – CLB (Configurable Logic Block), které jsou základem funkce hradlových polí. Právě jejich kon-



(a) Celkový pohled (b) Detail pravého horního rohu

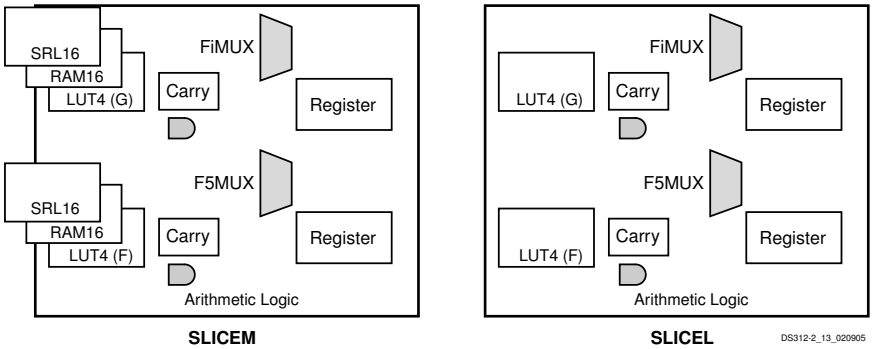
Obrázek 3.1: Vnitřní uspořádání FPGA z třídy Spartan-3A

figurováním a vzájemným propojováním jsou vytvářeny kombinační a sekvenční obvody podle našeho návrhu. Detail jedné CLB je na obrázku 3.2. Vidíme, že se dále dělí na 4 Slice (volně přeloženo jako „segment“), přičemž se rozlišují levé a pravé (funkčnost levých je rozšířena). Zjednodušené schéma obou typů je potom na obrázku 3.3. Na této úrovni se vyskytují nejmenší konfigurovatelné prvky. Jejich základní funkce budou vysvětleny následovně.



Obrázek 3.2: Náhled na jeden konfigurovatelný logický blok – CLB

Téměř nejdůležitějším prvkem je LUT4 (Look-Up Table 4-input), doslovně přeloženo jako „4-vstupová vyhledávací tabulka“. Umožňuje realizaci libovolné kombinační logické funkce čtyř proměnných (u součástek z vyšší řady nejsou výjimkou vícevstupové LUT). V principu se jedná o asynchronní paměť pouze pro čtení s velikostí 16 bitů. Zde se liší levý a pravý typ Slice: v levém typu lze do této paměti i synchronně zapisovat (pak se označuje jako RAM16), nebo je možné její obsah synchronně posouvat jako v posuvném registru, navíc ale zůstává možnost adresace jednotlivých bitů (pak se označuje jako SRL16).



Obrázek 3.3: Schematické znázornění nejmenších konfigurovatelných součástí uvnitř Slice. Vpravo je obyčejný typ, vlevo typ rozšířený.

Další klíčovou součástí je klopný obvod (Register). Může být řízen buď hranou (klopný obvod typu D), nebo úrovní. Navíc je vybaven vstupem povolující řídicí signál a rovněž disponuje vstupy pro nahození/shození výstupu, které mohou být volitelně synchronní či asynchronní. Z diagramu je patrné, že na každý blok LUT připadá jeden klopný obvod. Není tedy potřeba s nimi přehnaně šetřit. To umožňuje vyhnout se mnohaúrovňovým logickým řetězcům, které by svým velkým zpožděním omezovaly zbytek systém. Místo nich se používá proudové zpracování (pipeline) pracující na vyšší taktovací frekvenci a s vyšší celkové propustnosti.

Tyto dva bloky teoreticky postačují k implementaci libovolného kombinačního i sekvenčního obvodu. Aby ale jejich implementace a využití prostoru FPGA byly efektivnější, obsahují Slice další prvky. Především obvody pro rychlý přenos příznaku přetečení (Carry), které spojují vertikálně sousedící Slice a umožňují obecně zefektivnění realizace nejen aritmetických operací.

Dále se na několika úrovních vyskytují dedikované multiplexery, které především umožňují tvorbu rychlejších a prostorově úspornějších multiplexerů než s použitím samostatných LUT.

1. Multiplexer 1 z 4 – 1 Slice (2 LUT)
2. Multiplexer 1 z 16 – 1 CLB (8 LUT)
3. Multiplexer 1 z 32 – 2 CLB (16 LUT)

Jak vidíme, tak až do šířky 32 bitů se v multiplexeru neřadí LUT kaskádně a vychází 1 LUT na 2 multiplexované vstupy, což je dobré. I tak ale zabírají poměrně dost zdrojů a je Lepší se velkým multiplexerům při návrhu vyhýbat.

3.1.2 Ostatní bloky

Velmi důležitou součástí jsou IOB (Input/Output Block), tedy vstupně/výstupní bloky vyvedené na piny pouzdra. Jsou plně konfigurovatelné, podporují mnoho standardů (5 V logika již mezi ně běžně nepatří), mezi nimi i diferenciální přenos signálů, mají integrované zakončovací rezistory (samozřejmě také konfigurovatelné) a rovněž jimi lze jemně korigovat časová zpoždění signálů.

Jelikož v aplikacích s hradlovými poli je většinou vyžadováno poměrně hodně paměti a mnoho operací násobení, což jsou komponenty, které by implementované z LUT zabraly velkou část zdrojů, nehledě na výslednou rychlost, můžeme je v diagramu vidět jako zvláštní součásti. U řady Spartan-3 jsou dedikované násobičky v počtu jednotek až desítek. Jedná se o asynchronní násobičky s šířkou vstupů 18 bitů a výstupem šířky 48 bitů. V některých případech mohou být přímo vybaveny sčítačkou, případně dalšími obvody pro snadná zpracování signálů. Pamětem bude věnována celé část, prozatím zmiňme, že celková velikost pamětí se pohybuje od desítek po stovky kB.

Poslední, co můžeme vidět jsou bloky DCM (Digital Clock Manager) sloužící pro práci s hodinovými signály, jejich násobení, dělení, fázový posuv apod. Vyskytují se v počtu jednotek.

3.1.3 Propojovací sítě

Až propojovací sítě činí hradlová pole tím, čím jsou. Vhodně umístit a pospojovat CLB je pro návrhový systém náročný optimalizační úkol. Na jeho výsledku samozřejmě záleží konečná rychlost obvodu. Nevhodným návrhem/popisem logického obvodu, byť formálně správným, tedy můžeme výsledek poměrně dosti znehodnotit. Kromě propojovacích sítí pro obecné signály, s různou topologií, dosahem i zpožděním, obsahují hradlová pole také propojovací sítě globálního charakteru. Především se jedná o sítě hodinového signálu. Jejich počet je velmi omezený a kladou se na ně značné požadavky – musí rozvést hodinový signál ke všem hradlům s maximálně stejným zpožděním, aby se předešlo logickým hazardům. Přestože klopné obvody umožňují také vstup hodinového signálu z propojovací sítě pro obecné signály, není dobré tuto možnost využívat. V návrhu by obecně nemělo být více hodinových signálů, než je možné propojit globální sítí.

3.2 Vývojové nástroje

Základem pro popis logických obvodů jsou jazyky označované jako HDL (Hardware Description Language). K nejznámějším a nejrozšířenějším patří Verilog a VHDL, ve kterém byla vyvíjena tato práce a kterému bude věnována samostatná část. Oba popisují hardware na relativně nízké úrovni. Existují ale i nástroje a jazyky zaměřené na vyšší úroveň abstrakce, např. Simulink HDL Coder firmy MathWorks nebo jazyk Handel-C.

Xilinx nabízí integrované vývojové prostředí ISE Design Suite. Základní verze WebPack je dostupná zdarma, nepodporuje bohužel všechny součástky, ale omezení se týká pouze větších typů. Tato práce byla vyvíjena na prototypové desce s obvodem Virtex-2, který se už nevyrábí a nové nástroje ho zpětně nepodporují. Pro vývoj muselo být proto použito prostředí verze 9.2. Poznamenejme, že aktuální je nyní verze 13.1 (květen 2011). Následující tvrzení už tedy nemusí být aktuální.

Simulátor ve verzi 9.2. zřejmě nepodporuje jazyk VHDL úplně, protože nebylo možné přeložit některé testy. Byl proto použit open source simulační nástroj GHDL ve spojení s GTKWave pro zobrazení nasimulovaných průběhů. Rovněž editační schopnosti integrovaného vývojového prostředí nikterak nevynikají. K editaci je možné použít např. editor GNU Emacs jež nabízí množstvím nástrojů, které editaci usnadňují. S použitím nástroje GNU Make lze poté celý proces vývoje značně zautomatizovat. (Postup s GNU Make byl použit i během této práce. Nutno ale podotknout že vhodnějším nástrojem pro účely automatizace vývoje je jazyk tcl.)

3.2.1 Proces syntetizace

Následující odstavce popisují syntetizaci hardwaru a posloupnost kroků od zdrojových kódů jazyka HDL až po konfigurační soubor pro konkrétní součástku. Podrobnější informace lze nalézt v [16].

1. Všechny zdrojové soubory VHDL a Verilog se syntetizují nástrojem `xst`. Syntetizátor tyto soubory analyzuje a snaží se v nich rozpoznat zápisy stavových automatů, pamětí, registrů, multiplexerů, násobiček, sčítaček a dalších ucelenějších, komplikovanějších komponent. Následuje optimalizace rozpoznaných součástí (samozřejmě lze určit kritéria pro optimalizaci, jako jsou rychlost nebo velikost) a jejich převod do formy základních primitiv daného obvodu. Výstupem je soubor (příp. soubory) ve formátu `*.ngc`, který obsahuje ne jen popis hardwaru (netlist), ale i další dodatečné informace a omezení, které byla ve zdrojových souborech

zapsána. V tomto kroku nás nástroj upozorní na syntaktické chyby a především podá poměrně vyčerpávající zprávu o všech jeho krocích a komponentách, které v jednotlivých zdrojových souborech rozeznal. Můžeme si tedy ověřit, zda jsme naši myšlenku v jazyku HDL správně zapsali. Také poskytuje souhrnné informace o počtech logických bloků a předběžný odhad latencí a maximálních hodinových frekvencích.

Část výpisu pro jednu z entity vypadá např. následovně:

```
Synthesizing Unit <wave_table>.
  Related source file is "msp_motion/pwm/wave_table.vhd".
  Found 2048x10-bit single-port RAM <Mram_ram> for signal <ram>.
  Found 10-bit register for signal <DAT_O>.
  Found 1-bit register for signal <stb_delayed>.
  Summary:
inferred   1 RAM(s).
inferred  11 D-type flip-flop(s).
Unit <wave_table> synthesized.
```

2. Nástrojem `ngdbuild` jsou soubory `*.ngc`, `*.ucf` a příp. další přeloženy do jednoho jediného souboru `*.ngd` (Native Generic Database). Soubor `*.ucf` (User Constraints File) v překladu „soubor s uživatelskými omezeními“ poskytuje nástrojům informace typu: na jaký pin vyvést jaký signál a v jakém standardu, informace o časování vstupních signálů (především hodinových), do kterých míst se mají namapovat některé entity a mnoho dalších.

3. Při mapování, ke kterému použijeme nástroj `map`, dochází k mírným úpravám a převodu na konkrétní typy bloků (nikoliv konkrétní místo) v daném FPGA. Výstupem je `*.ncd` (Native Circuit Description).

4. Následuje krok rozložení a propojení všech součástí nazývaný Place&Route a prováděný nástrojem `par`. Výstupem je taktéž `*.ncd` soubor, ale obsahující kompletní informace o všech prvcích včetně polohy.

5. Posledním krokem je vytvoření tzv. bitfile `*.bit` nástrojem `bitgen`. Jedná se o binární soubor s daty určenými pro přímou konfiguraci FPGA přes rozhraní JTAG, nebo pro nahrání do konfigurační paměti, odkud si je může FPGA po zapnutí přečíst samo.

3.3 Jazyk VHDL

Jazyk VHDL nabízí poměrně velké možnosti, daleko více, než umí současné nástroje syntetizovat. A pokud má být výsledek navíc efektivní a úsporný, je nutné se při psaní ještě více omezit. Možnosti jazyka tedy využijeme především při tvorbě testů. Jazyk jako takový zde nebude rozebírán. Popsány jsou spíše konstrukce, kterým je vhodné se vyhnout, a drobné tipy, jež je vhodné znát při seznamování se s obvody FPGA. Množství příkladů a dalších užitečných informací, ze kterých bylo čerpáno, je nejen v literatuře [8] a [11], ale také přímo v manuálu syntetizačního nástroje [17]. V mnoha technických zprávách (zajímavé a pro začátek užitečné jsou např. [6], [4], [5] a [3]) je také rozebráno, jak logické obvody implementovat co možná nejefektivněji a ušetřit si zbytečné komplikace.

3.3.1 Použití resetu

V učebnicích jazyka VHDL jsou často k vidění ukázky sekvenčních logických obvodů s asynchronním resetem (viz. výpis 3.1), nejspíše pouze jako ukázka možného zápisu. Čtenář ale může lehce dojít k názoru, že to je správný postup, podpořený myšlenkou, že se obvody nějak inicializovat musí, a vytvořit si tak špatné návyky. Dalším logickým krokem je resety ze všech komponent spojit a vyvést na pouzdro obvodu. Tento postup je ale dosti špatný hned z několika důvodů.

Výpis 3.1: Nevhodný popis sekvenčního logického obvodu inicializovaného asynchronně na hodnotu 1.

```
1   entity ff is
2     port(
3       clk   : in  std_logic;
4       reset : in  std_logic;
5       d     : in  std_logic;
6       q     : out std_logic);
7   end ff;
8
9   architecture Behavioral of ff is
10  begin
11
12    process (clk, reset) is
13    begin
14      if reset = '1' then
15        q <= '1';
16      elsif clk'event and clk = '1' then
17        q <= d;
18      end if;
19    end process;
20
21  end Behavioral;
```

Použití asynchronního resetu by mělo nastat pouze v ojedinělých případech a vždy být podloženo oprávněnými důvody a zvážením možných dopadů. Asynchronní signál obecně může přijít v jakékoliv části periody hodinového signálu a vznik hazardu je proto velmi pravděpodobný. Pokud tedy potřebujeme reset, tak používáme zásadně synchronní. Pokud reset přichází z vnějšku, tak musíme navíc před jeho použitím zajisti úpravu tvaru na kvalitní pulz minimálně se synchronizovanou sestupnou hranou.

V žádném případě ale není nutné tímto způsobem vytvářet síť resetovacího signálu, který by inicializoval všechny entity. (Jedná se o velké plýtváni zdroji, navíc velmi ovlivňuje a zhoršuje optimalizaci při implementaci návrhu z hlediska časování.) K tomuto účelu je vyhrazena zvláštní globální resetovací síť (GSR - Global Set/Reset). FPGA je po konfiguraci v přesně definovaném stavu, tzn. i všechny paměti a klopné obvody jsou patřičně inicializovány. Přivedením kladného pulzu do sítě GSR je znovu inicializován stav všech klopných obvodů, nikoliv pamětí. Lze tedy dokonce detekovat, že k resetu došlo. Avšak síť GSR je nutné budit z vnějšku obvodu. V návrhu je tato síť reprezentována portem primitivní entity (u obvodů třídy Virtex-2 to je např. entita `STARTUP_VIRTEX2`).

Ve výsledku tedy často použití dodatečného resetu není nutné. Rozumný návrh ukazuje výpis 3.2. Reset je synchronní, pro případ, že by snad mohl být někdy použit. Pokud použit nebude, syntetizátor by ho měl při optimalizaci odstranit. Počáteční stav je definován už při deklaraci signálu, tzn. registr je inicializován na hodnotu `'1'` ještě před samotným spuštěním obvodu.

Jako příklad uveďme, že během návrhu této práce bylo tímto postupem redukce resetů dosaženo zmenšení návrhu o 3.2 % LUT (z 2652 na 2566).

Výpis 3.2: Správnější návrh sekvenčního logického obvodu. Inicializace probíhá už při konfiguraci FPGA a reset je synchronní

```
1   entity ff is
2     port(
3       clk   : in  std_logic;
4       reset : in  std_logic;
5       d     : in  std_logic;
6       q     : out std_logic);
7   end ff;
8
9   architecture Behavioral of ff is
10    signal q_inner : std_logic := '1';
11  begin
12
13    q <= q_inner;
14
15    process (clk) is
16    begin
```

```
17        if clk'event and clk = '1' then
18          if reset = '1' then
19            q_inner <= '1';
20          else
21            q_inner <= d;
22          end if;
23        end if;
24      end process;
25
26    end Behavioral;
```

3.3.2 Nedefinované stavy

Při práci bychom měli mít stále na paměti, že jazyk VHDL je syntetizačním nástrojem interpretován doslovně. Tj. co mu explicitně nepřikážeme, tak neprovede. Problém nastává, pokud např. nepopíšeme všechny stavy vstupů, které mohou nastat. Pravděpodobně nás nepopsané stavy vstupů nezajímají a náš kód je formálně správně. O tom, že nás nezajímají ale musíme informovat i syntetizátor. V opačném případě oprávněně předpokládá, že se při přechodu do nepopsaného stavu vstupů nesmí stát nic. Tedy že se po něm chce, aby zachoval stávající hodnoty výstupů. V důsledku dojde ke vzniku klopného obvodu řízeného úrovní (Latch), což není to, co požadujeme.

Porovnejme častou ukázku řešení multiplexeru ve výpisech 3.3 a 3.4. V případě, že nás hodnota jakéhokoli možného signálu v jistou chvíli nezajímá, přiřadíme mu úroveň `'X'`, která reprezentuje neznámý stav. Tím poskytujeme syntetizátoru větší prostor pro optimalizace, obvod zmenšíme, zrychlíme a vyhneme se možným problémům

Výpis 3.3: Multiplexer 1 ze 3. Pro vstup sel="11"je výstup nepopsaný a dochází tak k vytvoření klopného obvodu. (2 LUT + 1 registr)

```
1    q <= a when sel = "00" else
2         b when sel = "01" else
3         c when sel = "10";
```

Výpis 3.4: Správně popsaný multiplexer 1 ze 3. (1 LUT + 1 dedikovaný multiplexer)

```
1    q <= a when sel = "00" else
2         b when sel = "01" else
3         c when sel = "10" else
4         'X';
```

3.4 Paměti

Paměti jsou častou a důležitou součástí návrhu pro obvody FPGA. Existuje několik možností jak paměti vytvářet a inicializovat. V následujících částech nabízené možnosti rozebereme.

3.4.1 Druhy pamětí

Základní rozdělení pamětí dostupných v FPGA je na tzv. distribuované a blokové RAM.

Distribuované paměti jsou tvořeny z rozšířených LUT (označovaných jako RAM16, viz 3.3). Velikost základního elementu je tedy 16 bitů (1 x LUT4). Zápis probíhá synchronně, čtení asynchronně. Lze vytvořit i dvou-portové elementy distribuované RAM (použijí se dvě LUT4), přičemž jeden z portů potom umí pouze číst. Hlavní výhodou je, že paměťové buňky mohou být umístěny téměř libovolně, tudíž dobře integrovány s ostatní logikou. Spolu s asynchronním čtení se tak stávají ideálním řešením pro malé, rychlé paměti, FIFO paměti apod. Jejich počet je navíc poměrně veliký, jedná se o každou druhou LUT.

Blokové RAM jsou dedikové, plnohodnotné, dvou-portové paměti (viz. obrázek 3.1). Běžná velikost jednoho bloku je 16 kb. V součtu mohou být v FPGA desítky až stovky kb. Pracují v plně synchronním režimu (na rozdíl od distribuovaných pamětí). Adresní logika je konfigurovatelná, je tedy možné v určitých krocích měnit poměr mezi šířkou datové a adresové sběrnice. Oba porty jsou shodné a mohou zároveň zapisovat na shodné místo v paměti, přičemž lze nastavit prioritu zápisu pro jeden z nich.

Je také možné jeden blok použít jako dvě menší jedno-portové RAM, nebo ho použít jako více pamětí ROM bitovým rozdělením adresových a datových sběrnic. Další zajímavé použití je jako komplexní kombinační obvod s registry na výstupu nebo jako prostředek pro implementaci rozsáhlých stavových automatů, což za nás může udělat přímo syntetizátor logiky (zde je čtenář odkázán na manuál k atributům `bram_map` a `fsm_style`). Paměti složené z blokových RAM mohou být inicializovány v kódu VHDL (tudíž při syntetizaci), nebo již ve výsledné binárním konfiguračním souboru pomocí nástroje `data2mem`, což je zvláště výhodné při častých změnách dat, např. při použití syntetizovaných mikroprocesorů.

3.4.2 Způsoby vytvoření

Behaviorální popis

První možností jak vytvořit paměť je popsat funkci požadovaného typu paměti tzv. behaviorálně pomocí konstrukcí jazyka VHDL. Toto řešení je hardwarově nezávislé. Inicializační data lze popsat obdobným způsobem, jako v případě 3.2, ale taktéž lze k inicializaci využít plných možností jazyka VHDL a inicializovat data pomocí funkce, která do paměti předpočítá nějaké hodnoty podle funkčního předpisu, nebo je přečte ze souboru. Je ale prakticky nemožné tyto inicializační data změnit bez nového provedení syntetizace.

Syntetizační nástroj se obvykle sám rozhodně, zda paměť realizuje jako distribuovanou nebo blokovou. Jedním z kritérií je velikost (např. pro Virtex-2 je hranice 512 bitů). Typ paměti lze také syntetizačnímu prostředku vnutit pomocí atributů `ram_style` a `rom_style`.

Behaviorálně lze samozřejmě popsat i paměti, které nelze realizovat ani jako distribuované ani blokové. Může se tak lehce stát, že výsledek bude velmi neefektivní až nerealizovatelný. Je proto dobré si ve výpisu syntetizačního nástroje zkontrolovat, zda jsme kódem skutečně popsali to, co požadujeme.

Instanciování blokových ram

Další možností je složi paměť přímo z primitivů (entit) jednotlivých blokových RAM podle našich požadavků na velikost a šířku slova Konečné paměti a přidat adresový dekodér. Tímto způsobem máme plnou kontrolu nad návrhem a všemi dostupnými parametry primitivů. Inicializace může být provedena v jazyce VHDL pomocí generických parametrů pamětí. Zajímavější možností je inicializace nástrojem `data2mem`.

Použití generátoru

Součástí vývojových nástrojů v základní verzi je i `coregen` – nástroj umožňující jednoduché a rychlé vytváření komponent různého typu pouze na základě specifikování několika jejich požadovaných parametrů. Mezi nabízené komponenty patří i blokové a distribuované paměti. Jejich inicializaci lze provést ze souboru `*.coe`, což je textový formát firmy Xilinx pro záznam koeficientů digitálních filtrů, obecných dat apod. Pokud si jako typ paměti vybereme blokovou, tak za nás nástroj udělá to, co můžeme udělat ručně (viz. instance blokových ram), a udělá to velmi dobře. Můžeme si např. zvolit optimalizaci na velikost. Nástroj v tomto případě poskládá paměťový prostor z blokových ram různých datových šířek tak, že dojde k jejich maximálnímu využití a zároveň minimalizování ad-

resového dekodéru. V opačném případě vytvoří paměť jako prostou matici blokových RAM.

Funkce tohoto nástroje bohužel postrádá dobrou dokumentaci a především neprodukuje soubory `*.bmm` (Block RAM Memory Map), čímž je pro nás v našem případě jen těžko použitelná, jak následovně uvidíme.

3.4.3 Nástroj data2mem

Tento nástroj umožňuje zpětně měnit v konečném konfiguračním souboru pro FPGA (`*.bit`) inicializační data blokových RAM. Je tedy velmi výhodným pomocníkem, pokud máme v paměti např. program pro procesor, který budeme často měnit, jako v našem případě.

Jedním z formátů inicializačních dat, které umí tento nástroj přečíst, jsou také soubory `*.elf` – tedy soubory s binárními daty, jež mohou být výstupem softwarových kompilátorů.

Aby nástroj poznal, které bity vstupního souboru odpovídají kterým bitům v jaké blokové RAM, musíme mu poskytnout soubor `*.bmm` (Block RAM Memory Map), jež všechny tyto informace shrnuje. Protože nástroj `coregen` tyto soubory k pamětem negeneruje a navíc blokové RAM pojmenovává těžko interpretovatelnými názvy, nemůže být použit. Je tedy nutné paměti vytvářet jinak a soubory `*.bmm` vytvářet ručně. (Jejich generování umožňují až placené nástroje.) K tomu může pomůže manuál [2].

3.5 Přehled součástek

Základ nabídky firmy Xilinx tvoří FPGA řady Virtex-4 ,Virtex-5 a Virtex-6 pro nejnáročnější aplikace a řady Spartan-3 (se všemi jejími verzemi) pro nízkonákladové až středně náročné aplikace. Dále řada Spartan-6, která je modernějším a výkonnějším následníkem řady Spartan-3 (na první pohled jsou největší změnou LUT s 6 vstupy).

Další třídy FPGA Virtex-7, Kintex-7 a Artix-7 jsou firmou Xilinx teprve připravované. Jejich uvedení na trh může trvat poměrně dlouhou dobu.

V současné době je pro naše účely pravděpodobně nejvíce vhodná platforma Spartan-3A („A“ značí Advanced – vylepšená verze původní řady Spartan-3). V následujících tabulce 3.1 můžeme vidět srovnání jejích zástupců.

Jinou připravovanou zajímavostí v portfoliu firmy Xilinx jsou obvody Zynq-7000.[12] Tyto obvody již nejsou označované jako FPGA, ale jako EPP (Extensible Processing Platform). Základ tvoří dvoujádrový procesor ARM Cortex-A9 s koprocesorem počítajícím

Tabulka 3.1: Přehled obvodů z třídy Starta-3A.

Obvod	Počet CLB	Počet Slice	Distribuované RAM	Blokové RAM	Dedikované násobičky	Počet DCM	Uživatelské vstupy/výstupy
XC3S50A	176	704	11 kb	54 kb	3	2	144
XC3S200A	448	1792	28 kb	288 kb	16	4	248
XC3S400A	896	3584	56 kb	360 kb	20	4	311
XC3S700A	1472	5888	92 kb	360 kb	20	8	372
XC3S1400A	2816	11264	176 kb	576 kb	32	8	502

v plovoucí řádové čárce a s dvojitou přesností. Hradlové pole je až součást podřízená tomuto procesoru a je plně v jeho režii. Přestože se jedná o velmi výkonné řešení, cena by měla začínat na 15$ (ovšem při velkých odběrech). Mohlo by se tedy jednat o zajímavou platformu i pro řídicí aplikace. Pro zájemce dodejme, že zkušební vzorky by měly být široce dostupné v první polovině roku 2012.

Kapitola 4

Syntetizovaný mikrokontrolér

Syntetizovanými mikrokontroléry se označují mikrokontroléry realizované přímo v prostoru hradlového pole stejně jako ostatní logické obvody. Mohou být distribuovány např. v podobě HDL kódu, takže je možné si implementaci takového mikrokontroléru prohlédnou (případně i upravit), nebo v podobě už kompilovaného souboru, tj. bez zdrojových souborů.

V současnosti existuje poměrně velké množství syntetizovaných mikrokontrolérů, ať už komerčních, volně šiřitelný, s uzavřeným či otevřeným kódem. O výhodách a nevýhodách obdobných řešení v oblasti softwaru se již v minulosti vedly a stále vedou dlouhé diskuze. V případě syntetizovaných mikrokontrolérů je situace mírně odlišná tím, že jsou komerční produkty často vázány na hradlová pole jeho výrobce (čemuž se nelze příliš divit). Případný přechod na hradlová pole jiného výrobce potom může být poměrně nákladný a zdlouhavý, případně až nereálný

Co se týče výhod a nevýhod syntetizovaných mikrokontrolérů vzhledem ke klasickým jednočipovým mikrokontrolérům, je situace poměrně složitá. V sériové výrobě spotřební elektroniky se se syntetizovanými mikropočítači coby náhradou klasických jednočipových mikrokontrolérů setkáme pravděpodobně velmi ojediněle. Důvodem je především neekonomičnost takového řešení, protože prostor v hradlovém poli je vzhledem k cenám dnešních mikrokontrolérů relativně dosti drahý. Navíc větší prostor v hradlovém poli s sebou nese potřebu větší konfigurační paměti, které jsou taktéž poměrně drahé.

Mezi možné výhody patří, že je lze relativně jednoduše doplnit těsně vázanými a komplexními periferiemi, což může v některých případech implementaci zařízení velmi usnadnit a vlastnosti výsledného systému vylepšit. Taktéž při vývoji nemusíme ihned navrhovat vlastní prototypovou desku s klasickým mikrokontrolérem a přidruženým hradlovým polem. Z principu taktéž nehrozí ukončení výroby mikrokontroléru.

Běžnější je se setkat v hradlovém poli s malými syntetizovanými mikrokontroléry jako s podpůrnými prvky. A může jich zde být i několik. Pokud ale vyžadujeme plnohodnotný mikropočítač, jeví se dnes jako výhodnější možnost dvou-čipové řešení (pomineme-li některé velice specifické aplikace), tj. oddělené hradlové pole a jednočipový mikropočítač. Nabídka jednočipových mikropočítačů je dnes široká a ceny bývají velmi příznivé. Navíc již často obsahují rozhraní jako CAN, USB, Ethernet apod.

4.1 Výběr syntetizovaného mikrokontroléru

Z důvodů zmíněných v předchozích odstavcích a především z důvodu dostupnosti byl syntetizovaný mikrokontrolér vybírán z databáze projektu OpenCores. Na zkoušku byly zvoleny dva mikrokontroléry jakožto zástupci populárních syntetizovaných mikrokontrolérů v této komunitě:

- OpenMSP430 (`http://opencores.com/project,openmsp430`)
- Plasma - most MIPS I™ opcodes (`http://opencores.com/project,plasma`)

Pro konečnou implementaci byl vybrán openMSP430. Následuje krátké zhodnocení obou možností a odůvodnění výběru.

4.1.1 Plasma

Jedná se téměř o úplnou implementaci architektury MIPS I (implementovány nejsou pouze instrukce pro nezarovnaný přístup k paměti, které byly v době implementace chráněné patentem a v praxi pravděpodobně nejsou příliš potřebné). Následuje výčet klíčových vlastností:

- Šířka slova 32 bitů.
- Von Neumannova architektura.
- Hardwarová násobička/dělička. (cca 32 hodinových taktů)
- Hardwarová bitové posuvy. (pravděpodobně 1 hodinový takt)
- Podporuje chráněný režim.
- Pouze jeden vstup přerušení. (více přerušení nutno řešit softwarově nebo řadičem)
- Možnost blokování přístupu k paměti.

Vhodnější by tedy bylo označení mikropočítač než mikrokontrolér. Kromě samotného jádra mikropočítače projekt také obsahuje Ethernetový řadič, jednotku UART, vyrovnávací

paměť (cach) a řadič pamětí DDR. Všechny hardwarové části jsou popsány v jazyku VHDL. Dále projekt nabízí svůj vlastní operační systém reálného času Plasma RTOS a implementaci webového serveru. Dostupné jsou také další ukázkové programy jako např. test funkčnosti všech instrukcí.

Pro naše účely je toto řešení až zbytečně pokročilé a náročné na prostor v FPGA. V obvodu Virtex-2 si samotné jádro vyžádá přibližně 1500 jednotek Slice oproti přibližně 1000 Slice pro jádro openMSP430. Protože je Plasma 32-bitová architektura, jsou navíc větší i ostatní části jako periferie, paměti apod.

Ze subjektivního pohledu tento projekt taktéž nemá tak kvalitní dokumentaci a dostupnost informací jako openMSP430. Navíc je (opět subjektivně), celý projekt příliš provázaný s ukázkovou aplikací webového serveru, použitým vývojovým kitem a vlastním operačním systém. Úprava projektu a jeho reorganizace po softwarové i hardwarové stránce pro naše účely by si vyžádaly poměrně dosti času.

Přesto byly určité úpravy provedeny a mikropočítač na použité vývojové desce zprovozněn a připraven pro případné budoucí nasazení. Ukázkový projekt je součástí přiloženého CD. Shrnutí nejdůležitějších informací o této architektuře potřebných k případnému dalšímu vývoji můžeme najít např. v [1].

4.1.2 OpenMSP430

OpenMSP je implementací mikrokontroléru třídy MSP430 firmy Texas Instruments kompletně napsanou v jazyce Verilog. Jeho základní vlastnosti jsou následující:

- Šířka slova 16 bitů.
- Harvardská architektura (oddělené sběrnice pro program, data a periferie)
- Hardwarová násobička s akumulátorem (použita dedikovaná násobička v FPGA)
- 1 nemaskované a 14 maskovaných přerušení.

Bohužel projekt jako takový obsahuje minimum periferií. Jmenovitě TimerA, GPIO a již zmíněnou násobičku. Součástí projektu není ani hardwarová jednot UART. Naproti tomu je projekt velmi dobře dokumentován [7] a oproti mikropočítači Plasma je v mnohých směrech také úspornější. Tyto důvody a také rozšíření mikrokontrolérů msp430 vedly k jeho zvolení. Na rozdíl od mikropočítače Plasma taktéž nabízí i rozhraní pro debugger.

4.2 Použití mikrokontroléru openMSP430

Jádro syntetizovaného mikrokontroléru je nutné před použitím patřičně vybavit okolními komponentami, což popisuje následující text. Aby nebylo nutné tento postup vždy opakovat, nebo bezduše kopírovat, byl vytvořen modul `openmsp430`. Zde jsou připraveny kompletní konfigurace mikrokontroléru určené k okamžitému použití (včetně `*.bmm` souboru). Podrobnější informace najdeme uložené přímo v modulu na přiloženém CD.

4.2.1 Datová a programová sběrnice

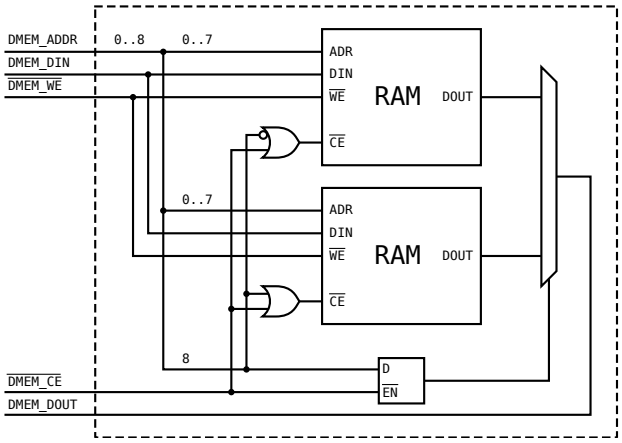
V první řadě je nutné k mikropočítači připojit programovou a datovou paměť. Možné postupy byly rozebrány v kapitole 3.4. Zvolen byl postup instanciováním blokových RAM. Za tímto účelem také byla napsána generická entita `ram_generic`, která pole blokových RAM tvoří automaticky stejně jako příslušný adresový dekodér a zároveň RAM smysluplně pojmenovává, takže je možné snadno napsat odpovídající `*.bmm` soubor a následně využít nástroj `data2mem`. Generická komponenta samozřejmě není nutná, ale pěkným způsobem ilustruje možnosti jazyka VHDL.

Bližší popis entity `ram_generic` je součástí jejího zdrojového kódu. Příklad, jak ji použít přináší připravené moduly mikrokontroléru. Způsob práce s nástrojem `data2mem` je potom možné najít v Makefile scriptu hlavního projektu.

Při práci s pamětmi je důležité vědět, že mikrokontrolér openMSP430 spoléhá na to, že data v datové i programové sběrnici se mění pouze při provedení operace čtení nebo zápisu. Jinak musí zůstat nezměněny. Nevědomost a nedodržení této podmínky způsobí nesprávné a těžko odhalitelné chování mikropočítače. Pokud tedy na sběrnici připojujeme více pamětí, je nutné stav výstupního multiplexeru měnit pouze při čtení nebo zápisu. Možné řešení ukazuje schéma 4.1. Zde je požadované funkcionality dosaženo klopným obvodem typu D zapojeným před multiplexer.

4.2.2 Periferní sběrnice

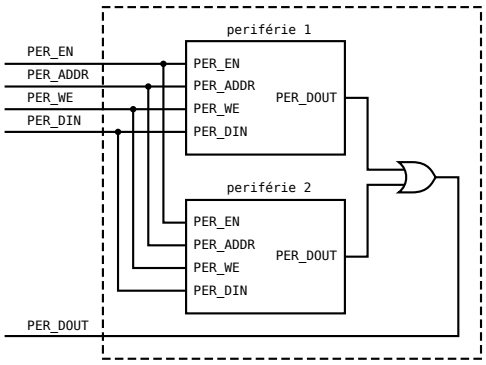
Sběrnice pro periferie je namapována do stejného logického adresového prostoru jako datová a programová sběrnice, ale podstatně se do nich liší svou velikostí (má pevnou šířku 256 slov) a především způsobem čtení. Čtení je totiž prováděno asynchronně. To znamená, že se na sběrnici musí objevit validní data ve stejném taktu, kdy se změní adresa, ne až při následující hraně hodinového signálu. Toto může být při implementaci periferií v některých případech překážkou. Ale z principu nic nebrání tomu, připojit periferie na



Obrázek 4.1: Způsob připojení více pamětí na jednu sběrnici (datovou, programovou) mikrokontroléru openMSP430.

datovou sběrnici, kde probíhá čtení synchronně.

Na obrázku 4.2 můžeme vidět doporučený postup jak vytvářet a připojovat periferie na periferní sběrnici mikrokontroléru openMSP430. Hlavní myšlenkou je, že každá periferie obsahuje úplný adresový dekodér a ten určí, zda je k ní přistupováno či nikoliv. Pokud ano, poskytuje na výstupu požadovaná data. Pokud k ní přistupováno není, tak udržuje na výstupu nízkou úroveň. Logický součet výstupních signálů všech periferií poté svým způsobem nahradí, jinak nutný, multiplexer.



Obrázek 4.2: Doporučovaný postup připojení periférií na periferní sběrnici.

Na úrovni sběrnice vypadá toto řešení jednoduše, ovšem o to více úsilí vyžaduje implementace samotných komponent. Navíc případná chyba v jedné komponentě může vést k nefunkčnosti všech. Taktéž je složitější sledovat, zda se paměťové prostory některých komponent nepřekrývají.

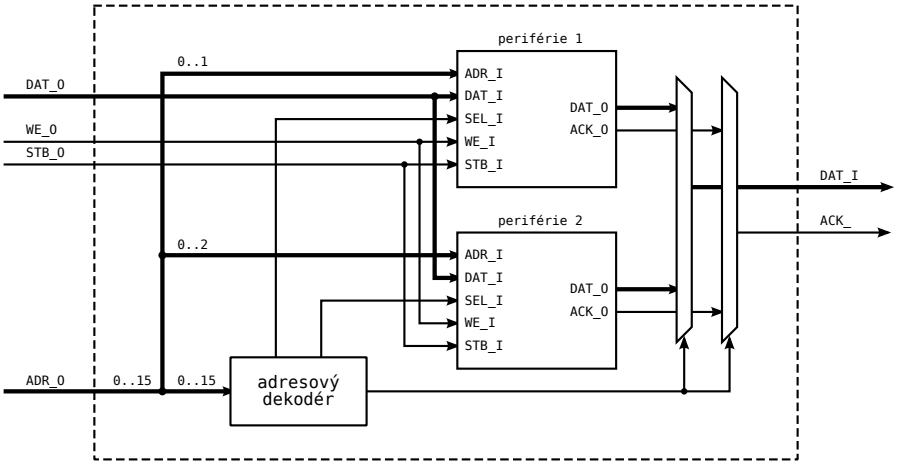
Z tohoto důvodu byl při realizaci sběrnice použit jiný přístup inspirovaný sběrnicí Wishbone, jak uvidíme v následující části.

4.2.3 Sběrnice Wishbone

Sběrnice Wishbone byla navržena pro co největší univerzálnost jak v přenosu dat, tak propojení uzlů a lze na ní vystavět systémy od těch nejjednodušších až po velmi komplexní s více uzly typu master. Úplnou specifikaci si můžeme přečíst v [10]. Zde se na funkci sběrnice podíváme pouze zběžně.

Propojení dvou periferií (uzly typu slave) je k vidění na schéma 4.3. První odlišností oproti řešení doporučovaném v projektu openMSP430 je realizace adresového dekodéru. Periférie dekóduje z adresy pouze tolik bitů, kolik odpovídá fyzické velikosti jejího adresového prostoru. Sama o sobě nemá informaci o tom, jak je nadřazený systém veliký a na jaké logické adrese se v něm skutečně nachází, což je zajisté výhodou. O interpretování celé adresy a adresování konkrétních periferií se stará adresový dekodér sběrnice. Činí tak nastavováním výstupů `SEL_O` (kód 1 z n). Taktéž multiplexuje výstupy periférií.

Operace čtení nebo zápisu se inicializuje nastavením signálu `STB_O` na vysokou úroveň. Mezi čtením a zápisem se rozhoduje úrovní signálu `WE_O`. Podle specifikace je operace dokončena, až když ji zařízení typu slave potvrdí nastavením signálu `ACK_O`.



Obrázek 4.3: Propojení dvou komponent typu slave ve sběrnici Wishbone.

Různé části tohoto konceptu byly při návrhu hojně uplatněny, ale celá specifikace byla z důvodu jednoduchosti dodržena málokdy. Většina navržených periferií tedy se sběrnicí Wishbone kompatibilních není.

Kapitola 5

Implementace hardwaru

Během této práce bylo v hardwaru implementováno řízení modifikovaným sinusovým signálem (viz. kapitola 2.2). Přičemž syntetizovaný hardware byl navrhován s důrazem na variabilitu, snadné a úsporné rozšíření pro řízení více os, celkovou výslednou velikost a především možnost ho relativně jednoduše rozšířit o momentové (proudové) řízení.

5.1 Rozhraní mikrokontroléru a regulační smyčky

Nutností je v první řadě propojit mikrokontrolér (ať už syntetizovaný nebo jakýkoliv jiný) s navrhovaným hardwarovým jádrem pro obsluhu bezkartáčových motorů a jeho řídicími registry. Od provedení této vazby se výrazně odvíjí následný způsob implementace samotného jádra i jeho klíčové vlastnosti.

5.1.1 Možné alternativy

Nabízí se tři možnosti: implementovat registry v podobě samostatných klopných obvodů, jako blokovou RAM nebo jako distribuovanou RAM.

Samostatné klopné obvody

Řešení pomocí samostatných klopných obvodů je poměrně přímočaré. Umožňuje ze strany jádra okamžitý přístup k hodnotám všech registrů. Bylo by tedy možné implementovat řídicí řetězec/řetězce paralelně a s využitím proudového zpracování. To ovšem v našem případě nepřináší žádnou výhodu, protože řídit bezkartáčový motor nevyžaduje tak velký výpočetní výkon. Naopak by toto řešení bylo velmi náročné na zdroje v hradlovém poli, stejně tak i následné připojení registrů do adresového prostoru mikropočítače.

Před každým jednotlivým registrem, do kterého bychom chtěli mít umožněn z mikrokontroléru zápis současně s jádrem, by musel být multiplexer vybírající 1 ze 2. A především bychom museli multiplexovat veškeré registry při čtení. Pokud by jedna osa byla ovládána 8 registry a osy byly celkem 4, tak bychom museli multiplexovat mezi 32 registry. Pokud by registry měly šířku 16 bitů, tak bychom pouze k implementaci multiplexeru na straně mikropočítače potřebovali $16 \cdot 16 = 256$ jednotek LUT4 (viz. sekce 3.1.1), což je přes 18 % velikosti nejmenšího obvodu řady Spartan-3A. A pokud by měl být jeden řídicí řetězec sdílen mezi více osami, musely by být obdobné multiplexery i na jeho straně. Navíc by mohly přijít komplikace při syntetizaci takto provázaného hardwaru. Toto řešení pro nás tedy není výhodné.

Distribuovaná RAM

Další možností by bylo přesunout registry do distribuovaných RAM. Došlo by ke zmenšení adresovacích multiplexerů, zároveň by v omezené míře mohlo fungovat proudové i paralelní zpracování, ale to je ale pro naše účely jen malým vylepšením.

Bloková RAM

Pro implementaci registrů řídicí smyčky tedy byla použita poslední varianta, která přináší největší úsporu prostoru – umístění veškerých registrů do blokové RAM. (Výjimku tvoří registry, které nejsou přímou součástí hardwarové smyčky, jako registry pro přímý přístup ke kvadraturnímu čítači nebo záchytné registry zachytávající jeho hodnotu, čas a další údaje při nějaké události. Ty z požadavku okamžité odezvy na nějakou událost být umístěny v blokové RAM nemohou.)

Protože jsou blokové RAM dvou-portové, můžeme jeden port vyhradit pro mikropočítač a druhý pro regulační smyčku. Dokonce lze blokovou RAM nastavit tak, že mikrokontrolér má prioritu při souběhu zápisů na stejnou fyzickou adresu (takto je v projektu sdílená RAM nastavena). Stále ale musíme mít na paměti, že problém souběhu tímto obecně není vyřešen, a musíme mu předcházet, pokud by k němu mohly nastat podmínky.

Netradiční implementace s blokovými RAM

Další alternativou je ponechat regulační smyčce prioritní přístup k oběma portům. To by zefektivnilo její implementaci, protože by měla umožněno pracovat se dvěma slovy zároveň. Jeden z portů by pak musel být s mikrokontrolérem sdílen. Protože by regulační

smyčka měla přednost, bylo by nutné realizovat operace čtení a zápisu mikrokontrolérem jako blokované, případně by si mikrokontrolér musel jiným způsobem ošetřit přístup k paměti. Což především ze strany mikropočítače návrh činí složitějším.

Zajímavější modifikací předchozí úvahy je použití dvou blokových RAM – jedné pro mikropočítač a jedné pro smyčku, která by tak mohla mít k dispozici oba porty. Jeden z nich by nebyl sdílen přímo s mikrokontrolérem, ale s jakýmsi dohlížecím obvodem, jež by patřičným způsobem obě paměti periodicky vzájemně zrcadlil a udržoval v nich konzistentní data. Jako příznaky aktuálnějších dat by mohly sloužit např. paritní bity přítomné ve větších blokových RAM. Toto řešení by se vyhlo blokovaným operacím hlavního mikrokontroléru (místo toho by byl blokován dohlížecí obvod) a zároveň by umožnilo úplnou kontrolu nad souběhem. Cenou za tyto výhody je doba propagace registrů mezi paměťmi a dále potřeba ještě jedné blokové RAM a dohlížecího obvodu, což by ale mělo být v porovnání s velikostmi ostatních komponent v systému téměř zanedbatelné.

5.1.2 Zvolené řešení

Poslední dvě rozebírané možnosti v části „Netradiční implementace s blokovými RAM“ jsou spíše zajímavým, možným návrhem do budoucna, jež by mohl přinést jisté výhody. Při implementaci této práce byl ale použit pro začátek jednodušší přístup, a sice přítomnost jedné paměti a vyhrazení samostatného portu jak mikrokontroléru, tak regulační smyčce.

Paměť je rozdělena na pole bloků velikosti 2^n, aby bylo možné jejich efektivní adresování, jak se později ukáže. Každý takový blok uchovává kompletní registry pro jednu osu. Mapování registrů v rámci jednoho bloku je uvedeno v tabulce 5.1. Poznamenejme, že celá regulační smyčka (entita `mcc`) pracuje se slovy šířky 16 bitů, tedy i sdílená paměť má tuto šířku.

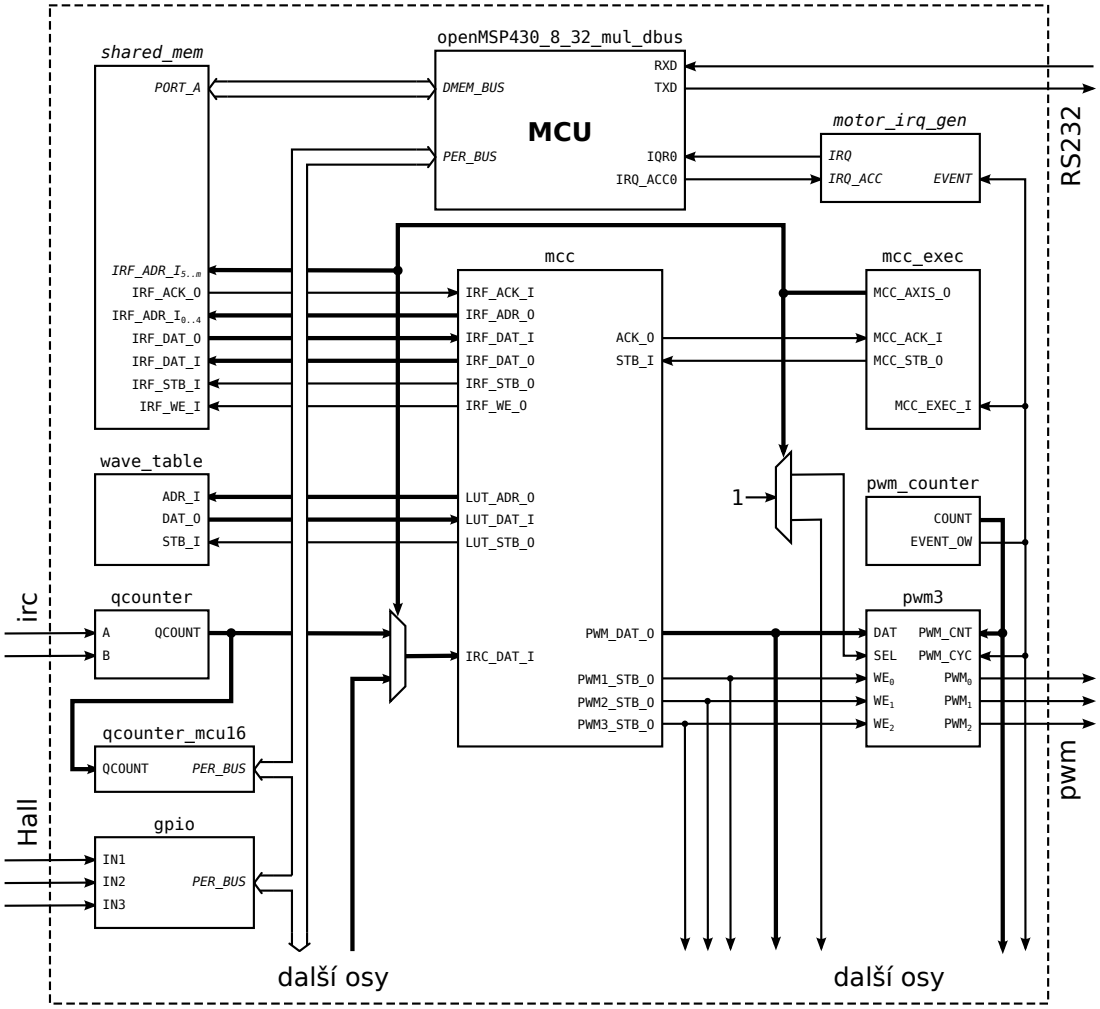
5.2 Implementace na nejvyšší úrovni

Implementaci na nejvyšší úrovni (tzv. top-level návrh) můžeme vidět v diagramu 5.1 a je taková, že regulační smyčka realizovaná entitou `mcc` je pouze jedna a postupně obsluhuje všechny osy/motory. Data všech os jsou v paměti sdílené s mikropočítačem, která je označena jako *`shared_mem`*. (V případě, že by požadovaný počet os nemohl být obsloužen jednou smyčkou, je samozřejmě možné smyčku i sdílenou paměť zdvojit a provádět řízení paralelně.)

Regulační smyčka `mcc` je postupně spouštěna pro všechny osy nadřazenou entitou

Tabulka 5.1: Mapa bloku ve sdílené paměti. Každý blok uchovává veškeré registry jedné osy. Všimněme si, že celý blok má velikost 32 slov (tedy 2^5) a že registry `MCC_Px` a `MCC_PWMx` (v budoucnu případně i další), které se vyskytují pro každou fázi, jsou zarovnány do bloků velikosti 4 slov (obecně 2^n). Důvodem je pozdější snadné adresování techto oblastí.

Offset	Název	Popis
0x00	MCC_EN	Příznaky povolující funkci jednotlivých bloků v regulační smyčce.
0x01	MCC_IRC	Hodnota kvadraturního čítače.
0x02	MCC_IBASE	Bázový registr, od kterého se odvíjí hodnota úhlu natočení (platí: MCC_ANGLE = MCC_IRC − MCC_IBASE).
0x03	MCC_IPER	Počet IRC kroků na jednu komutační periodu.
0x04	MCC_ANGLE	Úhel natočení rotoru (myšleno vzhledem ke komutaci) v jednotkách IRC čidla (platí: 0 ≤ MCC_ANGLE < MCC_IPER). Tento úhel značí směr kolmý ke směru magnetické indukce rotoru.
0x05	MCC_ACTION	Akční zásah, tj. velikost výsledného vektoru napětí (proudu). Uloženo ve formátu pevné desetinné čárky – 10 bitů za čárkou.
0x06	MCC_PWMMIN	Nejmenší hodnota PWM pro výpočet buzení modifikovanou sinusovkou.
⋮		
0x10	MCC_P1	Velikost jednotkového vektoru napětí promítnutého do směru první fáze.
0x11	MCC_PWM1	Akční zásah pro první fázi, tj. hodnota vynásobená MCC_ACTION přímo určená pro PWM generátor.
⋮		
0x14	MCC_P2	Obdobný registr jako MCC_P1, ale pro druhou fázi
0x15	MCC_PWM2	Obdobný registr jako MCC_PWM1, ale pro druhou fázi
⋮		
0x18	MCC_P3	Obdobný registr jako MCC_P1, ale pro třetí fázi
0x19	MCC_PWM3	Obdobný registr jako MCC_PWM1, ale pro třetí fázi
⋮		
0x1F		



Obrázek 5.1: Uspořádání komponent syntetizovaného hardwaru na nejvyšší úrovni – tzv. top-level návrh.

`mcc_exec`. Vykonávání této sekvence pro všechny osy je synchronizováno s PWM čítačem `pwm_counter`, který je společný pro celý návrh.

I blok *`irq_generator`* je synchronizovaný s PWM čítačem. Jednou za 22 period PWM signálu (tj. asi tisíckrát za sekundu) vyvolá v mikropočítači přerušení a spustí tak vykonání nadřazené softwarové regulační smyčky.

Další bloky jako samotné PWM generátory (`pwm3`), kvadraturní čítače (`qcounter`), obecné vstupy a výstupy (`gpio`), záchytné registry apod. musí být pro každou osu samostatné a nelze je jednoduše sdílet.

5.2.1 Entita mcc_exec

Nyní podrobněji rozebereme, jak entita `mcc_exec` řídí provádění hardwarové regulační smyčky `mcc`. Nejprve vždy nastaví signálem `MCC_AXIS_O` několik horních bitů adresy do

sdílené paměti na portu příslušejícímu regulační smyčce, čímž pro ni adresuje datovou oblast patřičné osy. Stejným signálem taktéž přepne jediné dva multiplexery viditelné na digramu – jedním je zvolen výstup odpovídajícího kvadraturního čítače a druhým je adresována příslušné skupina PWM generátorů pomocí k tomu určených signálů `SEL`. Komutační tabulka je taktéž připojena zvenčí a v případě, že ji vyžadujeme pro některé osy odlišnou, lze přidat další multiplexer a přepínat i mezi komutačními tabulkami. Tato možnost ale pravděpodobně nebude častá a není v digramu znázorněna.

Samotný běh regulační smyčky je řízen stylem dotaz a odpověď, jako komponenty na sběrnici Wishbone (viz. 4.2.3). Nadřazená entita `mcc_exec` vyšle dotaz/žádost nastavením signálu `MCC_STB_O` na vysokou úroveň a čeká, až přijde odpověď o splnění funkce v podobě vysoké úrovně na vstupu `MCC_ACK_I`. Poté postup opakuje pro další osu. Synchronizace začátku celého cyklu je realizována vstupem `MCC_EXEC_I`, který sleduje přetečení PWM čítače.

5.2.2 Ostatní entity

Následuje stručný popis několika dalších komponent na této úrovni návrhu:

wave_table

Jedná se o paměť uchovávající komutační tabulku. V současnosti je formát čísel bez znaménka a jejich rozsah je shodný s rozsahem PWM generátorů. Tabulka je inicializována externím souborem s daty. Vytvořit ho lze přiloženým skriptem `gen_sin_lut.m` programu Matlab nebo upraveným nástrojem `gen_phase_table` z knihovny PXMC.

qcounter

Kvadraturní čítač vyhodnocující signály z IRC senzorů. Poskytuje 32-bitovou informaci o poloze.

qcounter_mcu16

Tato entita slouží jako rozhraní mezi 32-bitovým výstupem kvadraturního čítače a 16-bitovou sběrnicí mikrokontroléru. Při čtení dolního slova čítače je jeho horní část uložena do registrů, takže je možné bezchybně přečíst celých 32 bitů. Není nutné tuto entitu duplikovat pro každý kvadraturní čítač, ale může být použita jako přizpůsobovací rozhraní k celé skupině 32-bitových komponent, ze kterých se pouze čte.

pwm_counter

Toto je obecně použitelný čítač (realizovaný entitou `counter`), od kterého se odvíjí generování všech PWM a spouštění softwarové i hardwarové regulační smyčky. Pokud požadujeme rozlišení připojených PWM generátorů na plných n bitů, musí být nastaveno resetování čítače při hodnotě $2^n - 2$.

pwm3

Obsahuje tři dílčí PWM generátory. Výstup každého generátoru je dán komparací uložené požadované úrovně a hodnoty externího čítače. Komponenta obsahuje mezipaměť, takže nově zapsaná hodnota se projeví až následující periodu PWM generátoru.

gpio

Jedná se o implementaci obecně použitelných vstupů a výstupů. Formálně obsahuje čtyři registry: registr pro čtení vstupů, nastavení výstupů a dva registry pro nahození/shození pouze některých výstupů zápisem odpovídající bitové masky.

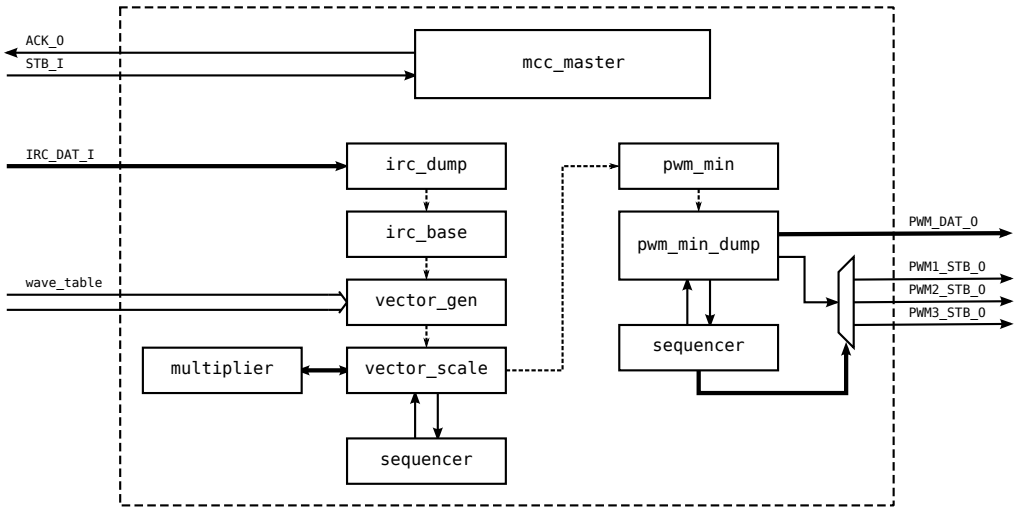
5.3 Regulační smyčka

Jádro regulační smyčky je implementováno v entitě `mcc` (Motion Control Chain), která víceméně pouze zapouzdřuje další entity, jež se procesu regulace přímo účastní. Schématické znázornění jejich vztahů je k vidění na obrázku 5.2.

Každá z vnořených entit je konečným stavovým automatem, který je schopný vykonat nad daty ve sdílené paměti operaci jemu příslušející. Vykonávání jednotlivých entit je řízeno entitou `mcc_master` obdobně, jako je entitou `mcc_exec` řízeno vykonávání regulační smyčky `mcc`.

Uvedené řešení má výhodu především v tom, že entity jsou vzájemně zcela izolované. Můžeme tedy problém dekomponovat na jednoduché kroky a každý takový krok implementovat samostatným stavovým automatem a k němu příslušejícími dalšími komponentami, které jsou potřebné, jako sčítačky, komparátory, bitové posuvy, násobičky apod. Z tohoto důvodu bylo toto řešení zvoleno. Další výhodu představuje, že lze takto jednotlivé komponenty snadno a odděleně testovat nejen pomocí simulátorů, ale také přímo v konečné aplikaci pomocí mikrokontroléru.

Nevýhodami je, že některé části, jako právě ony sčítačky, komparátory atd. jsou v podstatě duplicitní záležitostí a zbytečně zvyšují velikost výsledného hardwaru. Např.



Obrázek 5.2: Vazby entit uvnitř regulační smyčky `mcc`. Čárkované šipky značí pomyslnou cestu signálu (dat) od vstupu na výstup.

násobička – `multiplier` není součástí dílčí entity právě z důvodu případného sdílení s ostatními entitami. I když dnes už ani násobička není příliš velkou vzácností, jak ukazují její počty v tabulce 3.1. To ale neznamená, že nemůže být případně využita jinak a smysluplněji.

O mnoho větší nevýhodou je nutnost sdílet sběrnici k externí paměti (s registry) se všemi zapouzdřenými entitami, které se regulační smyčky účastní. Jak již bylo dříve vysvětleno, tento přístup vede k poměrně velkému počtu multiplexerů (především na datových a adresních vodičích) a není efektivní. Řešením by bylo použít centrální, sdílenou aritmeticko-logickou jednotku (ALU), přes níž by procházely všechna data. Multiplexovat by se poté musely pouze její řídicí signály.

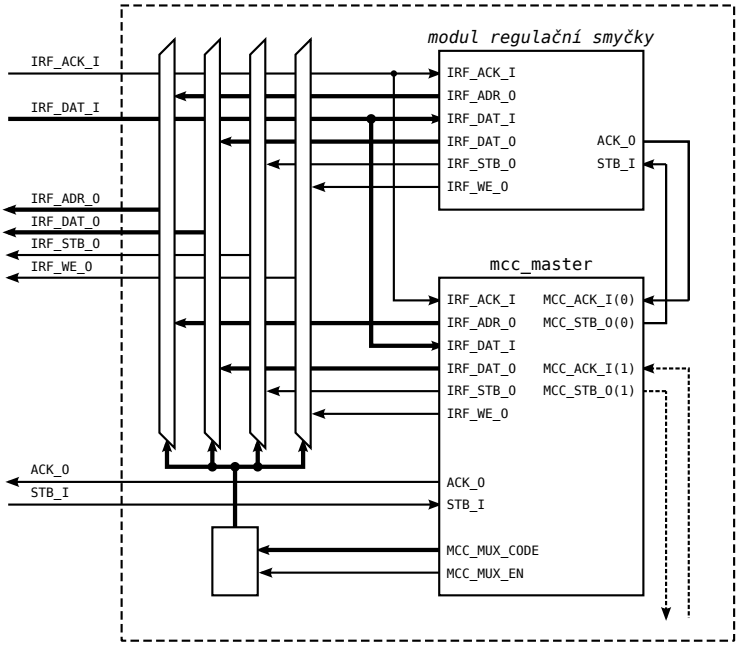
5.3.1 Entita mcc_master

Účelem entity `mcc_master` je řídit spouštění ostatních entit, které už přímo vykonávají svou část regulačního řetězce (viz. obrázek 5.2). Způsob připojení těchto entit k jednotce `mcc_master` ilustruje obrázek 5.3. Princi spouštění bloků pomocí signálů `STB` a `ACK` je obdobný jako ten popsaný v kapitole 5.2.1.

Rozšířením je, že lze vykonávání jednotlivých bloků regulační smyčky povolit/zakázat nastavením bitů v registru `MCC_EN`. Máme tedy možnost bez nutnosti rekonfigurace mírně měnit podobu regulační smyčky. Např. v případě potřeby je možné z mikrokontroléru přímo řídit hodnoty výstupního napětí na všech fázích (a využít, či nevyužít modifikovaného spínání), nebo řídit až velikost a úhel výstupního vektoru apod. V praxi je

nutností takto do regulačního řetězce zasahovat minimálně při inicializaci hardwaru.

Musíme ale počítat s tím, že pokud změnou v registru `MCC_EN` zakazujeme vykonávání některého bloku, změna se projeví až při následujícím regulačním cyklu. Ignorování této skutečnosti a okamžitá modifikace některého z dalších registrů pravděpodobně povede k souběhu a regulační smyčka upravená data přepíše. Řešením tohoto problému, jež by si vyžádalo nejmenší aktivitu na straně softwaru, by bylo použití dvou pamětí, jak bylo navrženo v části 5.1.1. Pokud jsou ale modifikované registry obnovované periodicky s každým cyklem softwarové regulační smyčky v mikrokontroléru, nemusí být tento problém pro funkčnost zásadní.



Obrázek 5.3: Princip propojení entity `mcc_master` a ostatních entit, jejichž vykonávání řídí.

5.3.2 Entita sequencer

Entita slouží k vykonání operace podřízené entity postupně nad všemi fázemi motoru. Pokud se podřízená entita snaží přistoupit k registrům první fáze, je tento stav detekován a přístup přesměrován do adresového prostoru právě zpracovávané fáze. Aby byl tento postup efektivní, jsou registry jednotlivých fází zarovnány do bloků velikosti 2^n.

5.3.3 Ostatní entity

Následuje stručný popis činnosti ostatních entit v regulační smyčce.

irc_dump

Uloží spodních 16 bitů hodnoty kvadraturního čítače do registru `MCC_IRC`.

irc_base

Pokud je to nutné, přičte nebo odečte registr `MCC_IPER` od `MCC_IBASE`. A to tak, aby platilo, že `MCC_ANGLE` = `MCC_IRC` − `MCC_IBASE` leží v intervalu 0 až `MCC_IPER` − 1 včetně. Do sdílené paměti taktéž uloží aktualizovanou hodnotu `MCC_ANGLE`. Pokud se registry `MCC_IRC` a `MCC_IBASE` liší o více než `MCC_IPER`, dojde ke splnění podmínky až po několika průchodech regulační smyčkou.

vector_gen

Rozloží jednotkový vektor s argumentem `MCC_ANGLE` do tří fází (tj. registrů `MCC_Px`). Rozklad je prováděn vyčítáním hodnot z komutační tabulky `wave_table`.

vector_scale

Vynásobí hodnoty v registrech `MCC_Px` velikostí akčního zásahu (velikostí napěťového vektoru) `MCC_ACTION`. Výsledek je uložen do odpovídajících registrů `MCC_PWMx`. Opakování pro všechny fáze je docíleno použitím entity `sequencer`. Akční zásah `MCC_ACTION` je v současnosti interpretován jako znaménkový typ s pevnou desetinnou čárkou na 10. místě.

pwm_min

Vyhledá v registrech `MCC_PWMx` nejmenší hodnotu PWM a uloží ji do registru `MCC_PWMMIN`.

pwm_min_dump

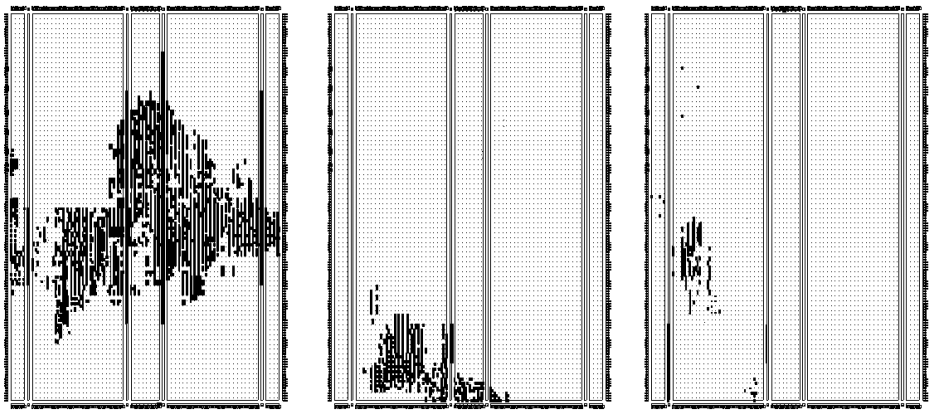
Nastavuje externí PWM generátory na hodnoty `MCC_PWMx` − `MCC_PWMMIN`, čímž je realizováno tzv. buzení modifikovanou sinusovkou. Opakování pro všechny fáze je docíleno použitím entity `sequencer`.

5.4 Vlastnosti výsledného návrhu

Simulací bylo změřeno, že vykonání celé hardwarové regulační smyčky trvá 78 hodinových taktů. PWM signál je generován s frekvencí 20 kHz a rozlišením 10 bitů. Na jeden cyklus

hardwarové smyčky tedy připadá více jak 1000 hodinových taktů. To znamená, že by neměl být problém navrženým hardwarem řídit s bohatou rezervou i 10 os, což je v praxi pro většinu aplikací více než dostačující.

Co se týče velikosti výsledného obvodu, tak na obrázku 5.4 můžeme hrubě porovnat, kolik prostoru která část zabírá. Orientační čísla v počtu jednotek LUT jsou následující:



Obrázek 5.4: Pohled z nástroje floorplan na využití prostoru hradlového pole. Obrázky popořadě zleva značí: 1. modul syntetizovaného mikrokontroléru (`openMSP430_8_32_mul_dbus`), 2. regulační smyčka (`mcc`), 3. všechno ostatní, tj. pwm generátory, vstupy, kvadraturní čítač atd.

Nejvíce zabírá syntetizovaný mikrokontrolér (včetně pamětí a modulu UART) – 2163. Samotné jádro mikrokontroléru o něco méně – 1953. Dále nejvíce zabírá hardwarová regulační smyčka `mcc` – 426. Všechny ostatní komponenty, tj. PWM generátory, kvadraturní čítač, vstupy, entita `mcc_exec` a další – pouze 162.

Celkově se jedná přibližně o 1380 jednotek Slice. Celý návrh by se tedy měl vejít do hradlového pole SC3S200A.

Poznamenejme, že multiplexery v `mcc`, které propojují jednotlivé moduly, obsazují 93 jednotek LUT, tj. přes 20 % celkové velikosti `mcc`.

Zde by pomohla již zmíněná implementace centrální aritmeticko logické jednotky. Byl učiněn pokus o její implementaci (zájemci ji mohou najít v odpovídajícím repositáři na přiloženém CD ve vývojové větvi `central_alu`). Přechodem modulu `irc_base` na tuto centrální ALU, která umí provádět operace sčítání, odčítání a násobení, se jeho velikost (včetně ALU) zmenšila ze 161 LUT na 109. Došlo ovšem také k jistému zpomalení, protože např. k porovnávání je místo dedikovaného komparátoru použito odčítání apod. Pokud by na centrální ALU přešly i ostatní moduly, mohla by být úspora prostoru markantní.

Kapitola 6

Implementace Softwaru

Mikrokontroléry MSP430 jsou poměrně rozšířené a existují pro ně dostupné nástroje. Jedním z nich je i překladač mspgcc (port známého překladače GNU GCC), který byl při vývoji použit. Výhodou je velmi dobře zpracovaný manuál [15]. Taktéž je dobré mít dispozici specifikaci architektury [14].

Za účelem řízení na úrovni mikrokontroléru bylo využito knihovny PXMC pro řízení motorů, frameworku Sysless a systému pro automatický překlad softwaru OMK (Ocera Make System). Knihovna PXMC byla vyvinuta ve firmě PiKRON a dále poskytnuta katedře řízení pro její výzkumné, studentské a open-source projekty s nabídkou na spolupráci na dalším vývoji. Vyčerpávající popis této knihovny můžeme nalézt v diplomové práci [13].

Nebude zde uveden podrobný postup zprovoznění překladače, doplnění základních knihoven pro použitý mikrokontrolér a následná portace knihovny PXMC. Omezíme se pouze na základní fakta a některé, ne obecně příliš známé, podrobnosti. Zájemci o tuto část práce naleznou její výsledky na přiloženém CD, konkrétně v modulech:

- Sysless (`sysless/arch/msp430/` a `sysless/board/msp430/`)
- PXMC (`pxmc/libs4c/pxmc_bsp/virtex2/`)

Dodejme, že komunikace s nadřazeným počítačem probíhá po sériové lince RS232. K interpretaci příkazů slouží knihovna cmdproc z frameworku Sysless.

6.1 Překlad programu

Jednou ze základních nutností je překladači sdělit, pro jakou verzi mikropočítače má překládat. Děje se tak klasicky přepínačem `-mmcu` (např. `-mmcu=msp430x423`). Při překladu

zvolíme verzi mikropočítače, který nejlépe odpovídá použité konfiguraci syntetizovaného mikropočítače. Liší se především přítomností hardwarové násobičky a další periférií a velikostí pamětí.

Snadno se ale může stát, že naše konfigurace pamětí žádné vyráběné verzi neodpovídá. V tom případě vybereme tu, která ji jinak odpovídá nejlépe, a musíme přikročit k modifikaci tzv. linker skriptu (např. `msp430x423.x`). Tento skript podává překladači (linkeru) informaci o tom ,jak má být výsledný program sestaven. Obsahuje tedy i informace o velikostech pamětí apod.

Velikosti datových oblastí upravíme následujícími řádky („text“ značí programovou paměť, „data“ paměť datovou):

```
text   (rx)       : ORIGIN = 0x8000,     LENGTH = 0x7fe0 /* 32K */
data   (rwx)      : ORIGIN = 0x0200,     LENGTH = 0x2000 /*  8K */
```

Dále je nutné upravit inicializační hodnotu ukazatele na zásobník. Ta by měla u mikrokontrolérů MSP430 vždy ukazovat na nejvyšší místo v datové paměti. Klasickým způsobem bychom požadované změny dosáhly následujícím řádkem:

```
PROVIDE (__stack = 0x2200) ;
```

Je ovšem možné velice snadno na tuto druhou úpravu zapomenout. Řešením je použít místo toho následující úpravu, která počítá inicializační hodnotu ukazatele `__stack` automaticky:

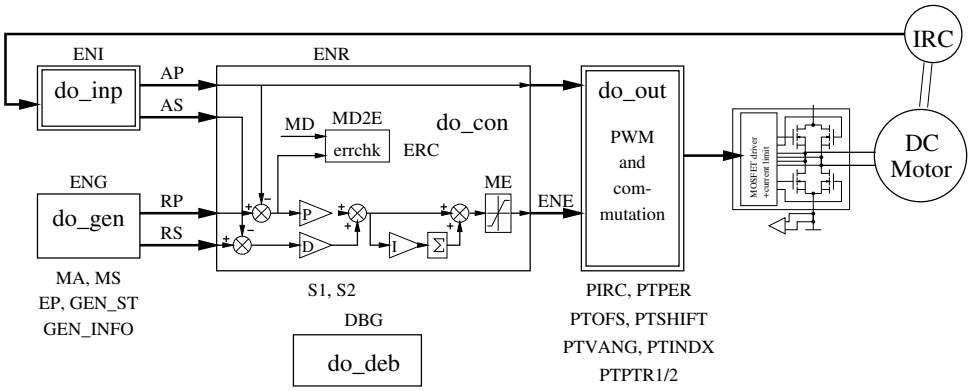
```
PROVIDE (__stack = ORIGIN(data) + LENGTH(data)) ;
```

Možnost druhého řešení se objevuje až v novějších verzích překladače a není k vidění příliš často, možná z důvodů zpětné kompatibility. Pokud otázku zpětné kompatibility nemusíme řešit, může nám ulehčit práci a čas strávený hledáním případné chyby.

6.1.1 Knihovna PXMC

Při portování knihovny PXMC je nutné především napsat funkce zajišťující rozhraní s novým hardwarem. Obrázek 6.1 ukazuje vnitřní strukturu knihovny při řízení jedné osy. Portování se týká především dvojitě orámovaných bloků pro vstup a výstup. Jejich podstatná část byla implementována v hradlovém poli. Zbylá část těchto dvou funkcí tvoří styčnou plochu mezi softwarem a hardwarem a řeší otázky okolo inicializace, když na začátku při spuštění systému není známa přesná poloha rotoru podle IRC snímače. Po

proběhnutí inicializace se vykonávaný kód výstupní funkce prakticky redukuje na pouhý zápis akčního zásahu do registru `MCC_ACTION` navrženého hardwaru – komutace již nemusí být řešena softwarově.



Obrázek 6.1: Struktura řízení jedné osy v knihovně PXMC. Dvojitě orámované bloky jsou z větší části realizovány v navrženém hardwaru.

V mikrokontroléru běží především jádro knihovny PXMC tvořené vlastní regulační smyčkou a generátorem tras.

Aby bylo možné k navrženému hardwaru přistupovat z jazyka C, byly napsány hlavičkové soubory definující logické adresy registrů hardwaru a poskytující makra, případně funkce, pro komunikaci s ním. Jmenovitě se jedná o soubory `gpio.h`, `mcc.h`, `quadcount.h` pro stejnojmenné entity napsané v jazyku VHDL.

6.1.2 Generátor komutačních profilů

Dříve byla komutace řešena softwarově právě ve výstupní funkci. A tabulka s komutačním profilem byla uložena v paměti mikrokontroléru. Pro její snadné generování je součástí knihovny PXMC nástroj `gen_phase_table`, který vygeneruje tabulku v podobě kódu jazyka C. Nyní je tabulka pevnou součástí hardwaru (viz. kapitola 5.2, entita `wave_table`) a soubor s komutačním profilem se jí předává jako generický parametr. Každý bod profilu musí být zapsán na zvláštním řádku a v binárním formátu (snadné zpracování jazykem VHDL). O generování těchto inicializačních souborů byl původní nástroj rozšířen.

Pro současnou implementaci hardwaru využijeme zejména spuštění nástroje s parametrem `-f vhdl-bias`, což způsobí generování čísel jako typu bez znaménka v rozsahu hodnot generátoru PWM (komutační profil je posunut směrem nahoru o polovinu rozsahu). Pro budoucí použití je připraveno i generování čísel jako typu se znaménkem – parametr `-f vhdl-sig`.

Kapitola 7

Závěr

Během práce se podařilo navrhnout funkční a použitelnou řídicí jednotku pro bezkartáčové motory kompletně implementovanou v hradlovém poli. Komutace probíhá modifikovaným sinusovým signálem. Řídicí jednotka tedy využije většiny možností, které bezkartáčové motory nabízejí, jako je kvalitní polohová regulace, hladký chod a další související výhody.

Přestože zařízení nebylo testováno na víceosém stroji, je pro provoz více os dobře připraveno. Taktéž vnitřní uspořádání syntetizovaného hardwaru umožňuje postupné rozšiřování a vylepšování jeho funkčnosti. Některé návrhy na budoucí zlepšení byly v textu zmíněny. Především se ale jedná o implementaci proudových regulátorů a následného momentového řízení, které bude pravděpodobně středem pozornosti při dalším vývoji.

Literatura

[1] MIPS-6371 Processor Specification. 2002.
URL: ⟨http://6004.csail.mit.edu/6.371/handouts/mips6371.pdf⟩

[2] *Data2MEM User Guide*. 2007.
URL: ⟨www.xilinx.com/itp/xilinx10/books/docs/d2m/d2m.pdf⟩

[3] Chapman, K.: Get your Priorities Right - Make your Design Up to 50% Smaller. Technická Zpráva WP275, Xilinx, 2007.
URL: ⟨http://www.xilinx.com/support/documentation/white_papers/wp275.pdf⟩

[4] Chapman, K.: Get Smart About Reset: Think Local, Not Global. Technická Zpráva WP272, Xilinx, 2008.
URL: ⟨http://www.xilinx.com/support/documentation/white_papers/wp272.pdf⟩

[5] Chapman, K.: Multiplexer Selection. Technická Zpráva WP274, Xilinx, 2008.
URL: ⟨http://www.xilinx.com/support/documentation/white_papers/wp274.pdf⟩

[6] Chapman, K.: Saving Costs with the SRL16E. Technická Zpráva WP271, Xilinx, 2008.
URL: ⟨http://www.xilinx.com/support/documentation/white_papers/wp271.pdf⟩

[7] Girard, O.: *openMSP430 an MSP430 clone*. 2010.
URL: ⟨http://opencores.org/usercontent,doc,1299010945⟩

[8] Hwang, E. O.: *Digital logic and microprocessor design with VHDL*. Victoria: Thomson, 2006, ISBN 0534465935.

[9] maxon motor: *Program 2010/11*.
URL: ⟨http://www.maxonmotor.ch/e-paper/⟩

[10] OpenCores: *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. 2010.
URL: ⟨http://cdn.opencores.org/downloads/wbspec_b4.pdf⟩

[11] Pedroni, V. A.: *Digital electronics and design with VHDL*. Amsterdam; Boston: Morgan Kaufmann, 2008, ISBN 978-0123742704.

[12] Santarini, M.: Zynq-7000 EPP Sets Stage for New Era of Innovations. *Xcell journal*, , č. 75, 2011.
URL: ⟨`www.xilinx.com/publications/archives/xcell/Xcell75.pdf`⟩

[13] Skup, K. R.: *Motion Control for Mobile Robots*. Diplomová práce, CTU in Prague, 2007.
URL: ⟨`http://support.dce.felk.cvut.cz/mediawiki/images/8/83/Dp_2007_skup_konrad.pdf`⟩

[14] Texas Instruments, Incorporated: *MSP430x1xx Family User's Guide (Rev. F)*. 2006.
URL: ⟨`http://focus.ti.com/lit/ug/slau049f/slau049f.pdf`⟩

[15] Underwood, S.: *A port of GNU tools to the Texas Instruments MSP430 microcontrollers*. 2003.
URL: ⟨`http://sourceforge.net/projects/mspgcc/files/documentation/031127/mspgcc-manual-20031127.pdf/download`⟩

[16] Xilinx: *Development System Reference Guide*. 2007.
URL: ⟨`http://www.xilinx.com/itp/xilinx10/books/docs/dev/dev.pdf`⟩

[17] Xilinx: *XST User Guide*. 2008.
URL: ⟨`www.xilinx.com/itp/xilinx9/books/docs/xst/xst.pdf`⟩

[18] Xilinx: *Spartan-3 FPGA Family*. 2009.
URL: ⟨`http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`⟩

Příloha A

Použité softwarové projekty

K implementaci práce byly použity následující softwarové projekty:

- Knihovna PXMC (PiKRON)
- Framework Sysless (Agrosoft, DCE FEL ČVUT a PiKRON)
- OMK – Ocera Make System (Projekt OCERA)
- Quadcount (Ing. Marek Peca)
- Syntetizovaný mikrokontrolér openMSP430 (opencores.com)
- Syntetizovaný mikrokontrolér Plasma (opencores.com)

Příloha B

Obsah přiloženého CD

`text/`	– Text bakalářské práce.
`modules/`	– Repositáře komponent vzniklých/upravených při vývoji.
`projects/`	– Repositáře původních projektů.
`finals/`	– Konečné projekty určené pro kompilaci.
`README`	– Bližší popis obsahu CD.