VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

VZOROVÉ ÚLOHY PRO HRADLOVÁ POLE

SAMPLES OF EXAMPLES FOR CONFIGURABLE GATE ARRAY

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Jan Bajer**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Michal Bastl**

SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav: Ústav mechaniky těles, mechatroniky a biomechaniky
Student: **Jan Bajer**
Studijní program: Aplikované vědy v inženýrství
Studijní obor: Mechatronika
Vedoucí práce: **Ing. Michal Bastl**
Akademický rok: 2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Vzorové úlohy pro hradlová pole

Stručná charakteristika problematiky úkolu:

Úkolem práce je vypracování vzorových úloh pro FPGA (programovatelná logická pole) firem Altera/Intel, nebo Xilinx. Práce predpokládá zpracování typických úloh vzhledem k oboru mechatronika. Výstup práce by měl posloužit jako výukový materiál zájemcům o problematiku.

Cíle bakalářské práce:

1) V teoretické části popište problematiku hradlových polí. Uveďte stručně historii technologie a její postavení vzhledem k mikroprocesorové technice . Popište rozdíly obou technologii a vhodné aplikace pro hradlová pole.
2) Stanovte použité vývojové nástroje. Popište hardware a software, se kterým budete pracovat. Dbejte na edukativní formu práce.
3) Vytvořte sadu základních realizací pro hradlová pole. Například generování PWM, komunikační sběrnice, PID regulátor.
4) Sestavte vzorové úlohy a zhodnoťte výstup práce. Může se jednat o komunikaci s PC a řízení krokového a stejnosměrného motorku.

Seznam doporučené literatury:

KRÁL, Jiří. Řešené příklady ve VHDL: Hradlová pole FPGA pro začátečníky. Praha: BEN - technická literatura, 2010, 127 s. ISBN 978-807-3002-572.

ISERMANN, Rolf. Mechatronic systems: fundamentals. New York: Springer, 2003. ISBN 1852336935.

ZDRÁLEK, Jaroslav. Programovatelné logické prvky [online]. Ostrava: Vysoká škola báňská - Technická univerzita, 2007, 164 s. [cit. 2015-05-16]. ISBN 978-80-248-1502-2. Dostupné z: http://www.elearn.vsb.cz/archivcd/FEI/PLP/zdralek_PLP.pdf

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

prof. Ing. Jindřich Petruška, CSc.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan fakulty

Abstrakt

Tato práce se zabývá problematikou hradlových polí a jejich postavením vzhledem k mikroprocesorové technice. Cílem je představit práci s hradlovými poli na sadě základních realizací v rámci oboru mechatronika. Úlohy jsou zpracovány s využitím jazyka VHDL a jsou primárně určeny na zařízení od společnosti Altera / Intel.

Summary

This thesis introduces the issue of configurable gate arrays and their position with respect to microprocessor technology. The aim is to present work with gate arrays on a set of basic realizations within the field of mechatronics. The examples are processed using VHDL and they are primarily intended for Altera / Intel devices.

Klíčová slova

Hradlová pole, FPGA, Altera, Intel, Cyclone, VHDL, UART, PWM, stejnosměrný motor, PID regulátor, krokový motor, AD převodník

Keywords

Configurable gate array, FPGA, Altera, Intel, Cyclone, VHDL, UART, PWM, DC motor, PID controller, stepper motor, AD converter

BAJER, J. *Vzorové úlohy pro hradlová pole*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2020. 58 s. Vedoucí práce: Ing. Michal Bastl.

Čestně prohlašuji, že jsem bakalářskou práci *Vzorové úlohy pro hradlová pole* vypracoval samostatně pod vedením Ing. Michala Bastla, s použitím materiálů uvedených v seznamu literatury.

Jan Bajer

Děkuji svému vedoucímu Ing. Michalu Bastlovi za věnovaný čas, technické rady, trpělivost a svědomité vedení mé práce.

Jan Bajer

Obsah

1 Úvod 3
2 Teoretická část 4
2.1 Historie 4
2.2 FPGA vs. mikrokontroler 5
2.3 Využití hradlových polí 5
2.4 Architektura 5
2.5 Jazyk VHDL 7
2.6 Teoretické základy k vypracování úloh 7
2.6.1 UART 7
2.6.2 PWM 8
2.6.3 Rotační enkodér 10
2.6.4 Stejnosměrný motor 10
2.6.5 Regulátor 11
2.6.6 Krokový motor 12
3 Praktická část 15
3.1 Stanovené cíle práce 15
3.2 Hardware použitý k sestavení úloh 15
3.3 Použitý software 19
3.3.1 Vývojové prostředí 19
3.3.2 Vytvoření projektu 19
3.3.3 Sériový plotter 21
3.4 Návrh sériové komunikace s PC 22
3.4.1 Vyrovnávací paměť 22
3.4.2 Vysílač 26
3.4.3 Přijímač 28
3.4.4 Ukázkový projekt Echo 30
3.5 Generátor PWM 31
3.6 Enkodér 32
3.7 Regulace DC motoru 34
3.7.1 Návrh regulátoru 34
3.7.2 Nastavení regulátoru s vykreslením průběhu 36
3.7.3 Řízení regulovaného DC motoru přes UART 37

3.8 Krokový motor 38
3.9 Analogová váha 40
4 Závěr..42
Literatura 43
Seznam použitých zkratek..46
Seznam příloh...47
Přílohy na SD kartě 47

1 Úvod

Je tomu už skoro 40 let, co se na trhu objevila první programovatelná hradlová pole, a ačkoliv je tato technologie už obecně známá, v širších kruzích se netěší velké oblibě právě kvůli zdánlivě složitému použití. Hlavně kvůli hardware popisujícím jazykům, pomocí kterých se hradlová pole programují, se většina lidí od této technologie odklání a raději využije některý z široké nabídky mikrokontrolerů. V některých aplikacích však může využití hradlového pole oproti mikrokontroleru přinést značné výhody.

Cílem této bakalářské práce je představit technologii programovatelných hradlových polí, jejich přednosti a příklady aplikací. Bude stručně popsána historie vývoje a architektura klasických hradlových polí, stejně jako postup k jejich naprogramování. Práce je zaměřena na hradlová pole od společnosti Altera / Intel, ale veškeré informace jsou přenositelné na zařízení od ostatních společností, jako je třeba Xilinx. Součástí teoretické části je uvedení základních informací potřebných k vypracování následných vzorových úloh.

V praktické části bude představen hardware použitý pro zpracování úloh a vývojové prostředí. Bude řečeno, jak založit nový projekt pro konkrétní hradlové pole a jak do něj nahrát vytvořený návrh. V sérii několika vzorových úloh bude přestaven postup práce s hradlovými poli. Bude využito jak popisovacích jazyků, běžně používaných k programování hradlových polí, tak schématických zapojení z vytvořených dílčích bloků. Součástí bude i ukázka simulace vytvořených návrhů. Výsledkem bude sestavení větších projektů z vytvořených částí.

2 Teoretická část

2.1 Historie

První programovatelná hradlová pole (angl. *Field-Programmable Gate Array, FPGA*) spatřila světlo světa v roce 1984, kdy společnost Xilinx uvedla na trh čip XC2064 s pouhými 64 logickými bloky. Každý z nich obsahoval dvě třívstupové look-up tabulky. Celkový počet hradel pak byl menší než 1000. Tehdy nebyly označovány FPGA do doby, než toto označení o čtyři roky později zpopularizovala společnost Actel, která v roce 1990 uvedla na trh hradlové pole s největší kapacitou pod názvem Actel 1280, využívající technologii antifuse [1], díky které nebyla potřeba druhé matrice pro uložení programu. Nevýhodou této technologie však bylo to, že se čip dal naprogramovat pouze jednou.

Z počátku byla kapacita hradlových polí velmi malá, a proto i návrhy byly spíše jednoduché. Z toho důvodu nebyla velká potřeba automatické syntézy a velká část návrhu se musela provádět ručně, včetně propojování logických bloků. Bylo zbytečné uvažovat o automatizaci procesu na osobním počítači, neboť i automatizace návrhů ASIC probíhala na sálových počítačích. Značné rozdíly v architekturách znemožňovaly užití univerzální designových nástrojů. Prodejci tak začali vyvíjet vlastní EDA pro svá zařízení, což nakonec vedlo ke zlepšení jejich architektury [1]. Další výhodou bylo, že zákazníci nemuseli platit za software třetích stran a snížily se tím celkové náklady. Cena za vývoj softwaru byla zakomponována do ceny samotného čipu.

Časem došlo k tomu, že velikosti požadovaných návrhů začaly přesahovat kapacity jednotlivých hradlových polí a začaly se tvořit návrhy s vícero čipy. Tím však vznikla potřeba po softwaru, který by tyto vícečipové struktury podporoval. Automatizace mapování a celkového kompilačního procesu stále nebyla považována za nezbytnou.

Následující roky došlo ke značnému nárůstu kapacity a poklesu ceny hradlových polí. V roce 1992 vyráběný čip XC4010 obsahoval až 10 tisíc hradel [1]. V devadesátých letech narůstala potřeba automatické syntézy a mapování. Výrobci se tak zaměřili právě na automatizaci nástrojů. Došlo i na změnu v architektuře hradlových polí, aby mapování bylo snadněji proveditelné automatickými nástroji. Z pamětí typu EPROM, antifuse a Flash se začalo přecházet na přívětivější SRAM.

Začátkem nového tisíciletí se programovatelná hradlová pole stala běžnou součástí řídích systémů. Výroba se rozdělila na nižší a vyšší třídu modelů. Nižší třída se vyznačovala menší kapacitou, výkonem, a hlavně nižší cenou. Do této kategorie se řadí dnes běžně používané rodiny Spartan od Xilinx a Cyclone od Altery. Vyšší třída se pak vyznačovala vysokou kapacitou, výkonem a cenou [1].

2.2 FPGA vs. mikrokontroler

Programovatelné hradlové pole je typ integrovaného obvodu, který může být naprogramován koncovým uživatelem. Samotné hradlové pole bez nahraného programu je pouze mrtvý kus křemíku bez žádných funkcí. Právě v tomhle se hradlové pole značně liší od mikrokontroleru. Mikrokontroler v sobě od výroby obsahuje řadu funkcí a periferií, které se naprogramováním jen nastavují a zapínají.

Mikrokontrolery jsou jednodušší a rychlejší k použití, navíc je k nim dostupné obrovské množství knihoven. Zato u hradlových polí je potřeba většinu věcí vytvořit od základu. Na druhou stranu, na hradlovém poli se dá vytvořit řada integrovaných obvodů, jako je nejen mikrokontroler, ale dokonce i kompletní počítač. Z hlediska výkonu se samozřejmě nebude ani zdaleka rovnat soudobým osobním počítačům, ale na lepším čipu je možné sestavit některé populární kusy z konce dvacátého století.

Velkou výhodou hradlových polí oproti mikrokontrolerům je simultánní běh jednotlivých částí, například v případě přerušení pak není potřeba řešit dobu na jeho vyřešení, neboť nijak neovlivní běh ostatních funkcí programu. Krom toho běh hradlových polí není nutně závislý na taktovací frekvenci připojeného krystalu a dokážou pracovat mnohem rychleji. Další nespornou výhodou hradlových polí je možnost kompletního přeřazení vstupů a výstupů, které probíhá přímo v návrhu a není pak potřeba měnit zapojení hardwaru. Možnost přemapování některých pinů už sice novější mikrokontrolery mají, ale pořád ne v takovém rozsahu jako hradlová pole.

Jednou z největších překážek při programování hradlových polí je používaný programovací jazyk. Na rozdíl od mikrokontrolerů, které se programují sekvenčními programovacími jazyky, se k naprogramování hradlových polí používají tzv. popisovací jazyky, které se označují HDL (Hardware Description Language). Mezi nejběžnější jazyky používané k popisu návrhu hradlových polí patří VHDL a Verilog HDL.

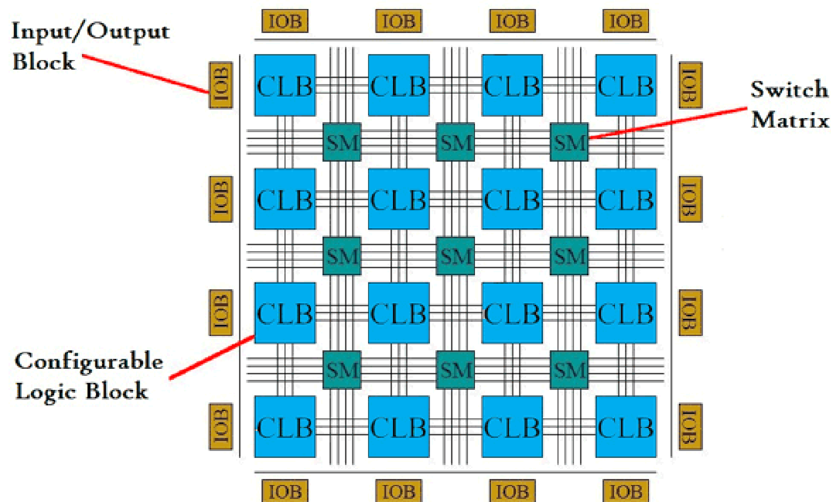
2.3 Využití hradlových polí

Programovatelná hradlové pole nalézají uplatnění všude, kde je potřeba rychle zpracovávat velké množství dat. Zpracování zvuku, obrazu, lékařské aplikace, letectví, obrana a kryptografie. Některá hradlová pole jsou určena přímo pro mise ve vesmíru. Díky schopnosti vykonávat až tisíce identických operací zároveň se v současné době čím dál více používají pro výpočty spojené s umělou inteligencí. Microsoft implementoval hradlová pole do svého datového centra k posílení vyhledávače Bing [22]. Hojně se jich také využitá při prototypování ASIC a nových jader procesorů [2].

2.4 Architektura

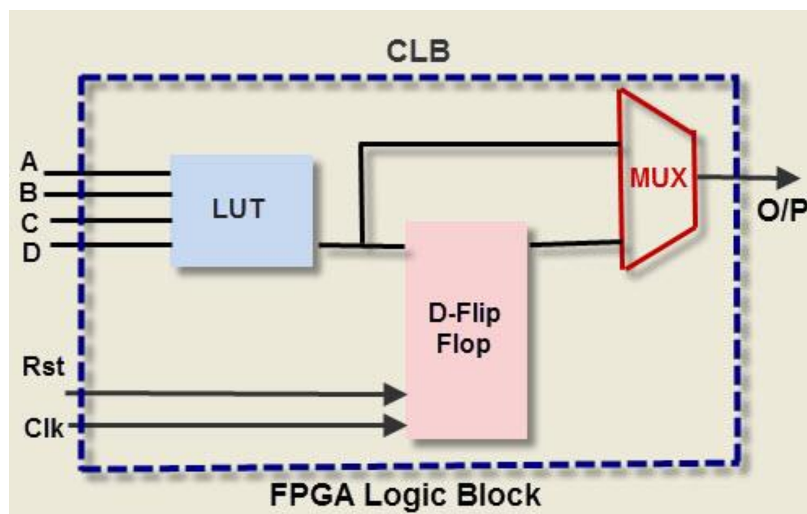
Hradlové pole se skládá ze tří hlavních částí – programovatelné logické bloky, které implementují logické funkce; programovatelná propojovací síť, která zajišťuje

vzájemnou interakci mezi jednotlivými bloky; a programovatelné vstupně-výstupní bloky (IOBs), které zajišťují připojení externích částí.



Obrázek 2.1: Blokové schéma hradlového pole [21]

Logické bloky realizují funkce požadované návrhem. Každý logický blok v sobě obsahuje několik dalších částí, mezi které patří look-up tabulky (LUTs), klopné obvody (D flip-flop) a multiplexy (MUXs) [3]. Look-up tabulky implementují kombinační logické funkce, klopné obvody uchovávají výstup z tabulky a multiplexy slouží k přepínaná signálů. Počet vstupních signálů do look-up tabulek se může pohybovat mezi 3, 4, 6 a v některých případech až 8 [2]. Dnešní tabulky mohou obsahovat i dva výstupní signály, každý pro jinou implementovanou funkci.



Obrázek 2.2: Programovatelný logický blok [25]

Hradlová pole obsahují tisíce až milióny těchto logických bloků a všechny dokážou pracovat paralelně. Jeden logický blok však nedokáže implementovat složitější funkce, a proto se pomocí vnitřní sítě spojují do komplexnějších zapojení.

Každý logický blok je připojen ke spínačové matici (angl. switch matrix), pomocí které se připojí k propojovací síti signálů procházející celým hradlovým polem. Matice obsahuje programovatelné multiplexy, které slouží k přepojování příslušných signálů k logickým blokům.

Vstupně-výstupní bloky slouží k interakci s okolním světem. Připojením externích součástí k vyvedeným pinům umožňuje propojit logické bloky s dalšími obvody.

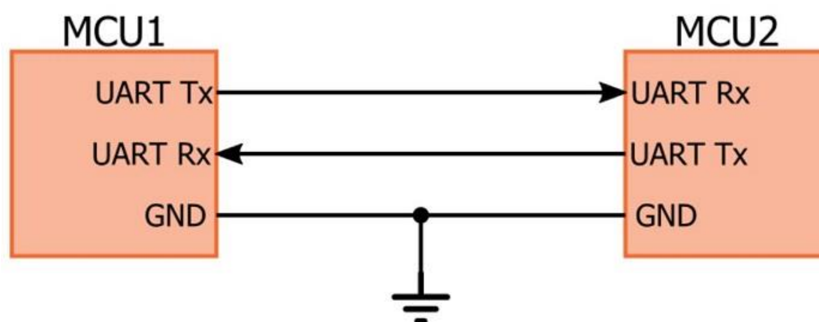
2.5 Jazyk VHDL

Jedním z jazyků používaných k programování hradlových polí je VHDL (VHSIC Hardware Description Language), které vzniklo v roce 1983 jako spin-off výzkumného programu VHSIC (Very High Speed Integrated Circuits) dotovaného Ministerstvem obrany Spojených států Amerických jako prostředek k dokumentaci chování zákaznických integrovaných obvodů (ASIC). Jedním z požadavků bylo, aby se syntaxe nového jazyka co nejvíce podobala syntaxi jazyku ADA, aby se vyhnulo opětovnému vývoji konceptů, které již byly úspěšně otestovány při vývoji tohoto jazyku [4]. Během VHSIC programu museli vývojáři čelit náročnému úkolu popsat neuvěřitelně komplexní obvody a řešit problémy s návrhem obvodů, které vyžadovaly několik výzkumných týmů. S využitím nástrojů jen na úrovni logických bran bylo brzy jasné, že je třeba vyvinout strukturovanější metody a nástroje. Konečný vývoj měl na starost tým inženýrů ze společností IBM, Texas Instruments a Intermetrics[5]. V roce 1985 byla představena první veřejná verze 7.2 jazyka VHDL. Dnes je nejpoužívanější verze z roku 1993.

2.6 Teoretické základy k vypracování úloh

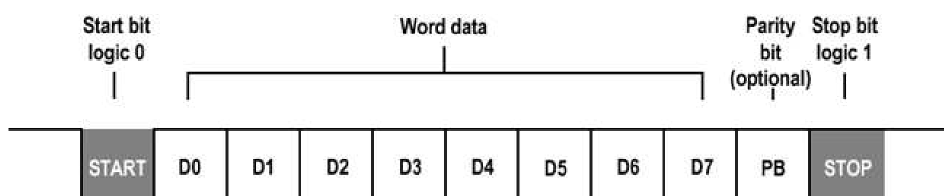
2.6.1 UART

Sériová komunikace UART (Universal Ansychronous Recceiver-Transmitter) je jednou ze základních sériových komunikací používaných v řídící elektronice a je běžně součástí mikrokontrolerů [7]. K přenosu dat využívá pouze dva datové signály pro obousměrnou komunikaci (full duplex). Pro komunikaci jedním směrem (simplex) stačí pouze jedna datová linka [6]. Nevýhodou však je, že narozdíl od ostatních běžně používaných protokolů, jako je SPI nebo I2C, může komunikace probíhat pouze mezi dvěma zařízeními.



Obrázek 2.3: Schématické zobrazení dvou zařízení komunikujících přes UART [6]

Komunikace probíhá v jednotlivých rámcích (angl. *frame*) pro každý byte. Přenos je zahájen START bitem, který představuje logická 0. Následuje 5 až 9 datových bitů (v dnešní době běžně 8) od nejnižšího po nejvyšší. Někdy za datovými bity následuje bit parity, který slouží jako jednoduchá kontrola úspěšného přenosu dat. Parita může být sudá nebo lichá. Sudá parita je nastavena na log 1, pokud je v přenášeném bytu lichý počet log 1. Stejně tak je lichá parita nastavena na log 1, pokud je v přenášeném bytu sudý počet. Jinými slovy, sudá parita zajišťuje, že datové bity a bit parity mají dohromady sudý počet logických 1, a analogicky lichá parita zajišťuje lichý počet [6]. Rámec je zakončen jedním nebo dvěma STOP bity, představovány logickou 1. Klidová hodnota signálu je taktéž reprezentována logickou 1. Jedná se o pozůstatek z dob využívání telegrafního přenosu, kdy přivedené napětí na komunikační lince ukazuje, že vysílací stanice není poškozená [7].



Obrázek 2.4: Komunikační protokol UART [19]

Rychlost přenosu dat bývá definována v tzv. baudech (angl. baud rate). V případě komunikace UART tato hodnota odpovídá bitům za sekundu (bps). Převrácená hodnota pak udává dobu přenosu jednoho bitu [6].

2.6.2 PWM

Pulsně šířková modulace (angl. Pulse Width Modulation) je metoda přenosu analogové hodnoty pomocí digitálního signálu [8]. Přenášená hodnota je reprezentována střední hodnotou signálu. Pro výpočet střední hodnoty $\bar{y}$ signálu popsaného funkcí $f(t)$ s periodou T platí vztah 2.1.

$$\bar{y} = \frac{1}{T}\int_{0}^{T} f(t)dt \tag{2.1}$$

Pro obdélníkový signál se spodní hodnotou y_0, horní hodnotou y_1 a se střídou s (angl. duty cycle), která je poměr doby horní hodnoty (zapnuto) k celkové periodě T, získáme vztah 2.2.

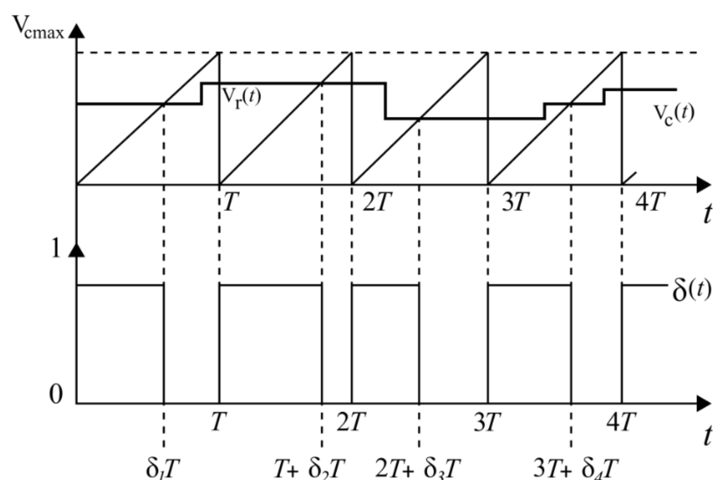
$$\bar{y} = \frac{1}{T}\int_{0}^{sT} y_1 dt + \frac{1}{T}\int_{sT}^{T} y_0 dt \tag{2.2}$$

Pro stejnosměrné nesymetrické napětí dosadíme za $y_0 = 0$ a $y_1 = U$, kde U je hodnota napětí horní hodnoty, získáme vztah pro střední hodnotu v závislosti na střídě a vstupním napětí.

$$\bar{y} = \frac{1}{T}(s \cdot T \cdot U) = s \cdot U \tag{2.3}$$

Z výsledného vztahu 2.3 je zřejmé, že velikost střední hodnoty je přímo úměrná střídě signálu.

Ke generování PWM se používá čítač/časovač (angl. counter/timer), který je běžnou periferií mikrokontrolerů. Často používaná rozlišení jsou 10 nebo 16 bitů. V závislosti na řídícím hodinovém signálu se periodicky zvyšuje (nebo snižuje) hodnota čítače. Čas kompletního zaplnění čítače udává periodu PWM signálu. Hodnota čítače se pak porovnává s referenční hodnotou, kterou chceme pomocí PWM přenést. Pokud je hodnota čítače menší než referenční hodnota, je na výstup PWM přiřazena log 1, a pokud je větší, tak je přiřazena log 0.



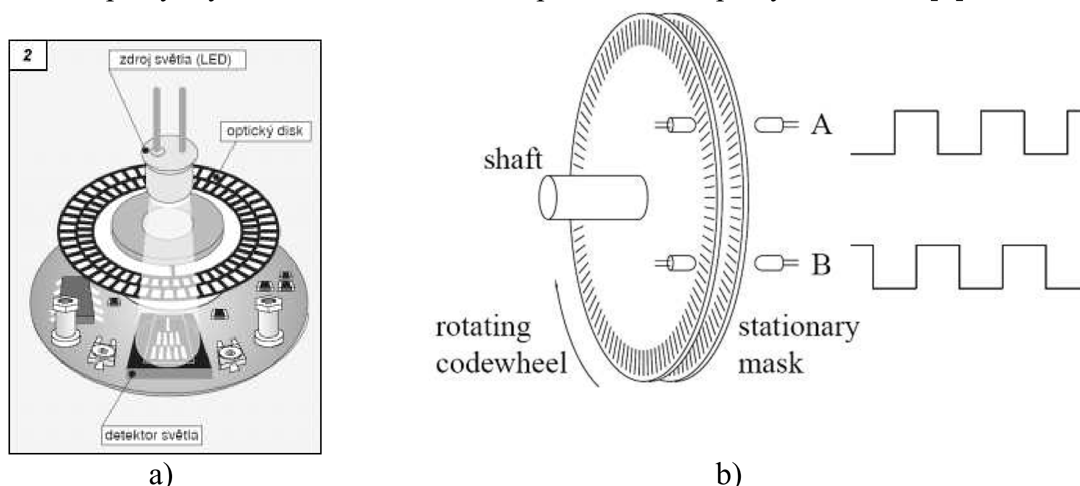
Obrázek 2.5: Princip generování PWM [31]

PWM se využívá například k nastavení jasu světel, řízení motoru, v kombinaci s dolnofrekvenční propustí získáme jednoduchý D/A převodník a může být použita u zesilovačů třídy D. Běžně se také využívá u DC/DC měničů a měničů frekvencí [8].

2.6.3 Rotační enkodér

Enkodéry slouží k elektromechanické přeměně rotačního pohybu na digitální signál, jehož zpracováním dokážeme určit nejen polohu natočení, ale také rychlost otáčení. Nejběžnějším využitím enkodérů je snímání rychlosti motorů za účelem regulace. Často se využívá také jako vstupní periferie nahrazující analogové potenciometry.

Jedním z typů používaných enkodérů je optický enkodér. Optický disk je spojen s hřídelí motoru. Světlo vytvářené zdrojem světla umístěným před diskem prochází průhlednými otvory v disku. Na druhé straně disku je proti zdroji světla umístěn optický snímač, který světelné impulsy, vytvářené otáčením disku, přetváří na impulsy elektrické [9].



Obrázek 2.6: a) Optický enkodér [27]; b) Inkrementální rotační enkodér [29]

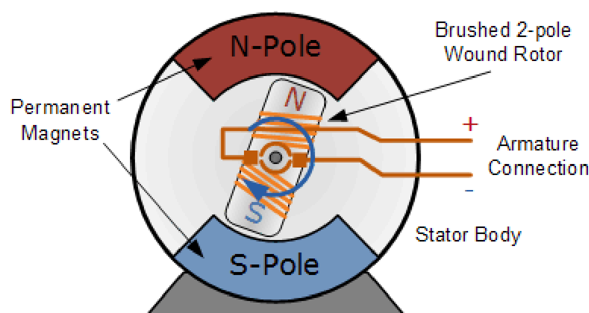
Magnetické enkodéry fungují podobně jako ty optické, ale místo světla využívají magnetická pole. Namísto optického disku je k hřídeli připevněno kolo s magnetickými póly, které se otáčí v blízkosti Hallovy sondy. Změnami magnetického pole dochází k vytváření elektrických impulsů, stejně jako u optických enkodérů [10].

Inkrementální rotační enkodéry většinou generují dva signály, označované jako *kanál A* a *kanál B*, někdy také jako *CLK* a *DATA*. Tyto signály jsou oproti sobě posunuté o 90 stupňů. Díky tomuto posuvu je pak možné určit nejen rychlost otáčení, ale také směr.

2.6.4 Stejnosměrný motor

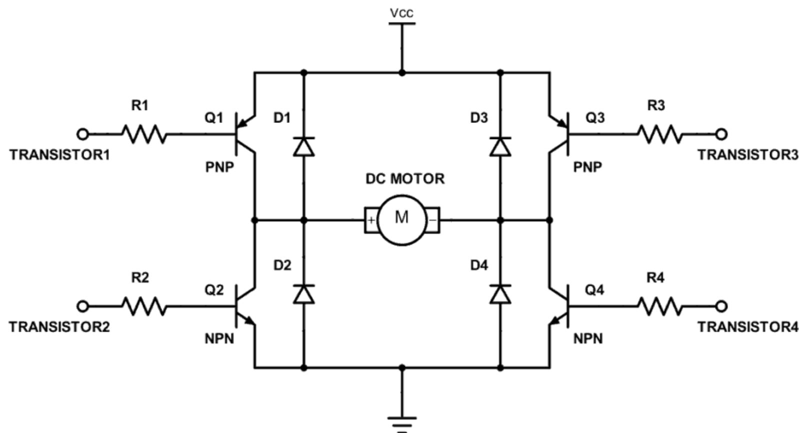
Stejnosměrný motor je jedním z nejstarších strojů k přeměně elektromechanické energie. Skládá se ze statoru a rotoru. V případě kartáčového motoru je rotor tvořen elektromagnetem, který se označuje jako kotva. Stator je tvořen elektromagnetem nebo permanentním magnetem [11]. V případě malých stejnosměrných motorů to bývá zpravidla permanentní magnet.

Rychlost otáček stejnosměrného motoru závisí na vzájemným pohybu dvou magnetických polí, jedno buzené permanentními magnety statoru, které je nepohyblivé, a druhé pohyblivé, které vytváří vinutí rotoru. Vzájemným přitahováním opačných pólů a odpuzováním stejných pólů vzniká točivý moment. Díky přepínání vinutí kotvy (tzv. komutaci) dochází k přepólování tak, aby se zachovával směr otáčení. Aby byl pohyb plynulejší, má kotva vícero vinutí [11].



Obrázek 2.7: Stejnosměrný motor s permanentním magnetem [24]

K řízení rychlosti a směru otáčení se používá tzv. H-můstek (angl. H-bridge), jehož název vychází ze schématického vzhledu zapojení tranzistorů. Jeho konstrukce umožňuje spínáním tranzistorů nejen ovládat rychlost, ale i otočit směr proudu procházející motorem a tím změnit směr otáčení.



Obrázek 2.8: Zapojení H-můstku [28]

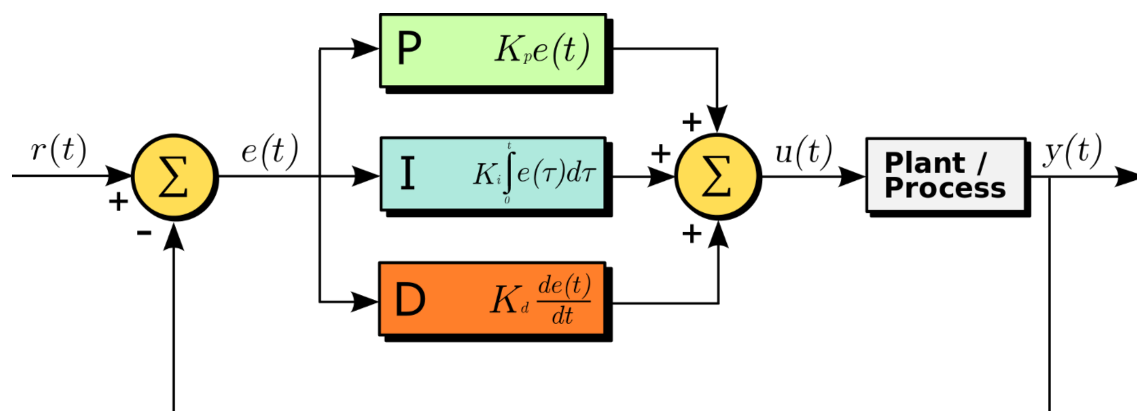
2.6.5 Regulátor

Regulace slouží k automatickým úpravám regulovaného systém bez ohledu na to, jak se změní jejich vnější podmínky. V případě regulace motoru se pak může jednat o udržování konstantních otáček bez ohledu na zatížení motoru.

Běžně používaným regulátorem je PID regulátor (proporcionálně integračně derivační). Často však bývá derivační složka vypnutá a je realizován pouze PI regulátor. Míru zásahu

jednotlivých složek regulátoru určují regulační konstanty K_p pro proporcionální složku, K_i pro integrační složku a K_d pro derivační složku.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \quad (2.4)$$



Obrázek 2.9: Blokové schéma PID regulátoru [23]

Proporcionální složka regulátoru je přímo úměrná odchylce skutečné hodnoty $y(t)$ od žádané hodnoty $r(t)$. Čím větší je aktuální odchylka $e(t)$, tím větší je i zásah do systému $u(t)$. Ačkoliv je tato část regulace nejjednodušší na realizaci, může snadno dojít k přesažení žádané hodnoty, což způsobí otočení smyslu zásahu regulace. Tímto způsobem může dojít ke kmitání kolem žádané hodnoty a nemusí dojít k ustálení [14].

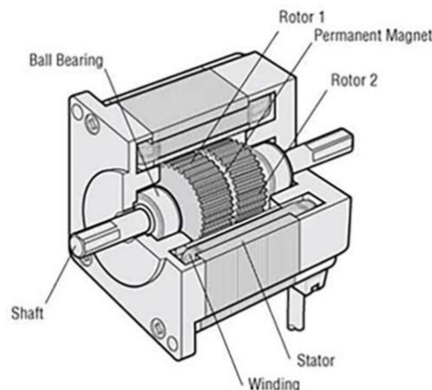
Integrační složka pracuje s integrálem regulační odchylky za celou dobu regulace. Díky tomu je velikost zásahu regulační složky závislá nejen na velikosti odchylky, ale také na době jejího trvání. Ve srovnání s proporcionální složkou je pomalejší, ale i malá odchylka se pak za určitou dobu projeví jako značný zásah do systému.

Derivační složka pracuje s rychlostí změny regulační odchylky. Čím je větší změna, tím je větší zásah. Protože však derivační složka nepracuje s odchylkou samotnou ale pouze s její změnou, nedokáže nikdy dosáhnout žádané hodnoty. Zásah derivační složky dokáže být velmi agresivní. Regulovaná soustava se pak může velmi snadno rozkmitat a stát se nestabilní. Z toho důvodu je v praxi filtrována.

2.6.6 Krokový motor

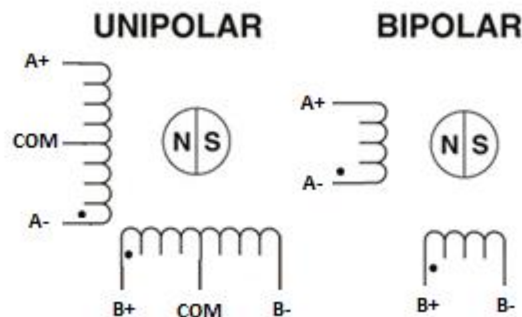
Krokové motory patří mezi synchronní motory napájené stejnosměrným proudem. Otáčení motoru je nespojité, což je při nízkých otáčkách snadno vidět. Motor se otáčí po jednotlivých krocích, které představují stabilní polohy. Jednou z hlavních charakteristik motoru je právě počet kroků na otáčku, nebo úhel mezi jednotlivými kroky. Vyznačují se vysokým kroutícím momentem ve stabilní poloze, který se označuje *holding torque* a definován jako velikost momentu potřebného k otočení o jeden krok při jmenovitém proudu a nulových otáčkách [12].

Stator krokového motoru je tvořen několika ozubenými elektromagnety okolo ozubeného rotoru z magnetického nebo feromagnetického materiálu. Napájení elektromagnetů je ovládáno řídící elektronikou. Otáčení rotoru je způsobeno postupným zapínáním elektromagnetů v určitém pořadí. Jakmile se zapne první elektromagnet, zuby rotoru a elektromagnetu se vlivem magnetického pole zarovnají. Další elektromagnet v pořadí je ale vůči zubům rotoru lehce posunutý. Když se pak vypne první elektromagnet a zapne se druhý, rotor se pohne, aby byly zuby rotoru a elektromagnetu zarovnány. Vykonaný pohyb je označován jako krok motoru [13]. Jednotlivé elektromagnety jsou rozděleny do skupin zvaných fáze (Obrázek 2.11). Postupným zapínáním a vypínáním jednotlivých elektromagnetů, resp. fází, dochází k pohybu rotoru po jednotlivých krocích.



Obrázek 2.10: Schématický řez hybridním krokovým motorem [30]

Podle typu vinutí budících cívek rozdělujeme krokové motory na unipolární a bipolární. Unipolární motor se pozná podle počtu vyvedených kontaktů, těch bývá 5 nebo 6. Vinutí fází unipolární motorů má centrální vývod, který může být pro obě fáze společný. Tohle zapojení umožňuje jednoduší řízení než u bipolárního motoru, protože není potřeba měnit směr proudu. K řízení unipolárního krokového motoru pak stačí jen 4 tranzistory [12].



Obrázek 2.11: Vinutí fází unipolárního a bipolárního krokového motoru [26]

Dalším rozdělením krokových motorů je podle buzení rotorů. Rozeznáváme pasivní, aktivní a hybridní točivé motory. Rotor pasivních motorů je sestaven z plechů z magneticky měkkého materiálu, který tvoří vyniklé póly. Aktivní motory mají rotor

tvořený permanentním magnetem. Kombinací těch dvou provedení získáme hybridní motor. Rotor hybridního motoru tvoří axiálně položený permanentní magnet a na jeho pólech jsou umístěné koruny z feromagnetického materiálu. Koruny jsou vzájemně natočené tak, aby se při axiálním pohledu zuby korun severního a jižního pólu střídaly [13].

Hybridní krokový motor má větší kroutící moment, menší kroky a vysokou efektivitu při nižších otáčkách. Díky permanentnímu magnetu dokáže držet polohu i bez napájení statorového vinutí. Na druhou stranu je oproti ostatním variantám těžší a dražší.

K ovládání krokového motoru je nutná řídící elektronika, která vytváří přesnou posloupnost pulsů, odpovídající požadovaným otáčkám a směru otáčení. K řízení bipolárního krokového motoru se dá využít, stejně jako u stejnosměrného motoru, H-můstek. Protože má krokový motor dvě fáze, jsou k jeho řízení potřeba dva můstky.

Krokové motory se využívají všude, kde je nutné přesné pozicování. Běžně se s nimi můžeme setkat v tiskárnách, skenerech, diskových mechanikách, optikách foťáků, robotice, CNC strojích a 3D tiskárnách [13].

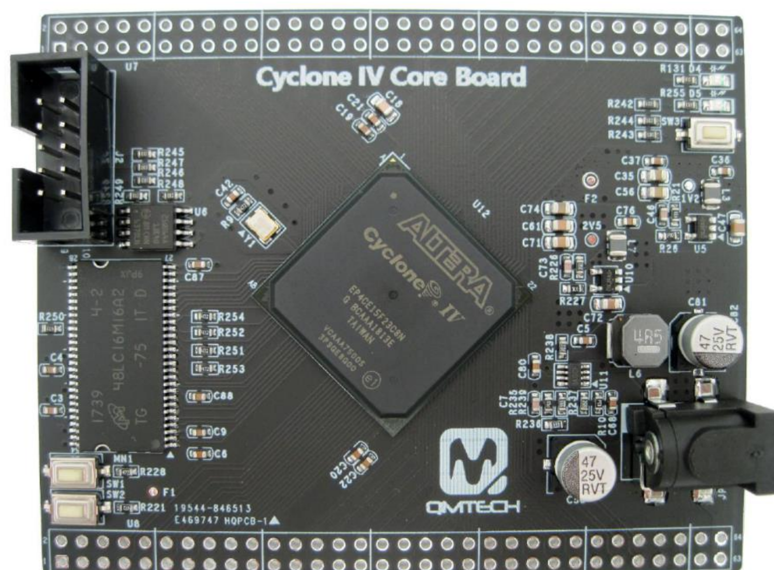
3 Praktická část

3.1 Stanovené cíle práce

- představení použitého hardwaru k sestavení vzorových úloh, vývojové desky a programátoru
- představení vývojového prostředí, stručný popis k práci potřebných částí a vytvoření nového projektu pro použitý hardware
- kompletní proces naprogramování hradlového pole ukázaný na prvním jednoduchém příkladu; zahrnuje vytvoření souboru, návrh chování, simulaci, přiřazení vstupních a výstupních pinů a nahrání do hradlového pole
- vytvoření sady základních realizací pro hradlové pole
- sestavení projektů využívajících základní úlohy

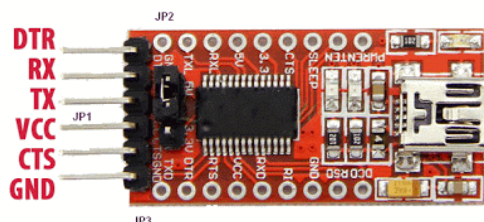
3.2 Hardware použitý k sestavení úloh

Použitá vývojová deska *Cyclone IV Core Board* od čínské společnost *QMTECH* obsahuje hradlové pole EP4CE15F23C8N od společnosti Altera, externí krystal s frekvencí 50 MHz, externí paměť SDRAM s kapacitou 32 MB a spínaný regulátor napětí na 3.3 voltů. Hradlové pole má vyvedených 108 vstupně-výstupných pinů (IOs), celkovou kapacitou 15 tisíc logických bloků a až 512 kB paměťových bloků typu RAM. Na desce jsou také dvě uživatelská tlačítka a jedna programovatelná světelná dioda. Druhá světelná dioda slouží jako indikátor napájení. K naprogramování hradlového pole je použit programátor *ALTERA USB Blaster*.



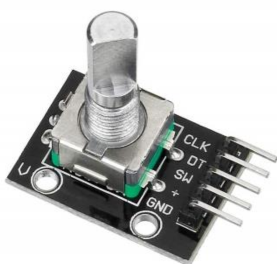
Obrázek 3.1: Vývojová deska od společnosti QMTECH s čipem EP4CE15F23C8N [32]

Pro převod sériových komunikací mezi protokoly USB a UART je použit modul s čipem FTDI232. Jeho velkou výhodou je možnost přepínání mezi 3.3 V a 5 V napěťovou úrovní.



Obrázek 3.2: USB/UART převodník [15]

Jako jedna ze vstupních periferií nahrazující často používaný analogový potenciometr je jednoduchý modul s mechanickým rotačním enkodérem s tlačítkem. K samotném enkodéru jsou na modulu již připojeny nezbytné pasivní součástky, a proto je modul možné zapojit přímo k vývojové desce.



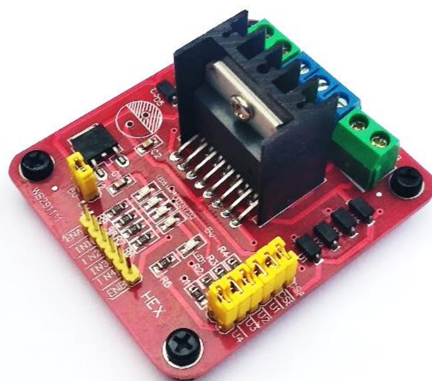
Obrázek 3.3: Mechanický rotační enkodér [33]

Prvním typem motoru, který je potřeba pro sestavení následujících úloh, je stejnosměrný motor. Jmenovité napětí motoru je 12 V. Motor je dodáván s převodovkou 10:1 a výsledné otáčky jsou 600 RPM. K motoru je připojen magnetický enkodér s Hallovými sondami, který má pro jednu otáčku hřídele 444 tiků.

a) b)

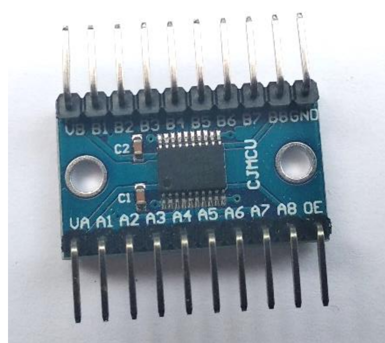
Obrázek 3.4: a) Stejnosměrný motor; b) Zapojení napájení motoru a enkodéru [34]

Integrovaný obvod L298N obsahuje dva kompletní H-můstky, kterými je možné individuálně ovládat dva stejnosměrné motory nebo jeden bipolární krokový motor. Řídící obvod pracuje na 5 V logice. Zdroj napětí 5 V může být externí, nebo se dá využít integrovaný lineární regulátor napětí na desce.



Obrázek 3.5: Modul L298N

Pro převod mezi napěťovými úrovněmi hradlového pole a H-můstku je použit modul s čipem TXS0108E, který umožňuje obousměrný převod úrovní na 8 signálech zároveň.

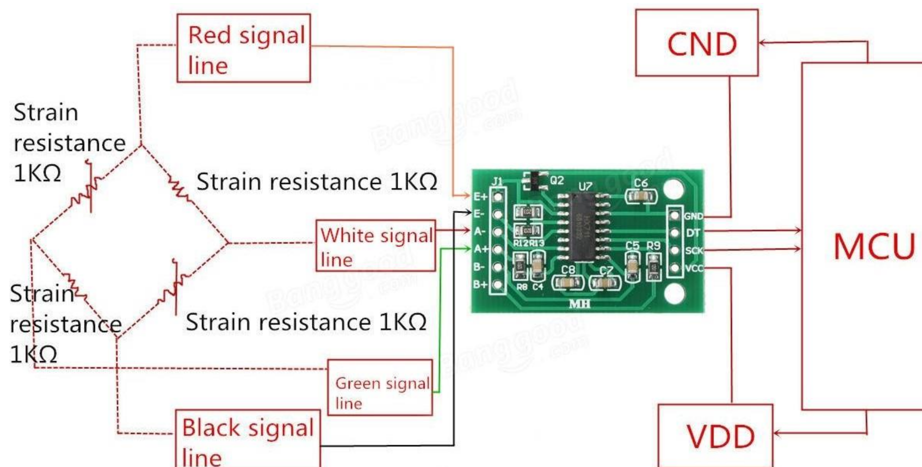


Obrázek 3.6: Převodník napěťových úrovní

Druhým typem motoru potřebným k sestavení úloh je krokový motor. Pro následující úlohy je použit bipolární hybridní krokový motor s označením 17HS3430 od značky HANPOSE. Počet kroků na jednu otáčku je 200, úhel kroku je tedy 1.8°. Jmenovité napětí motoru je 12 V.

Obrázek 3.7: Bipolární krokový motor 17HS3430

Poslední potřebnou součástí je analogově-digitální převodník. Zvolený modul používá 24bitový převodník HX711 s dvěma kanály a vnitřním programovatelným zesilovačem s maximálním zesílením 128. Vyznačuje se vysokou přesností měření analogového signálu. K převodníku je připojen vážící senzor dimenzovaný na 5 kg.

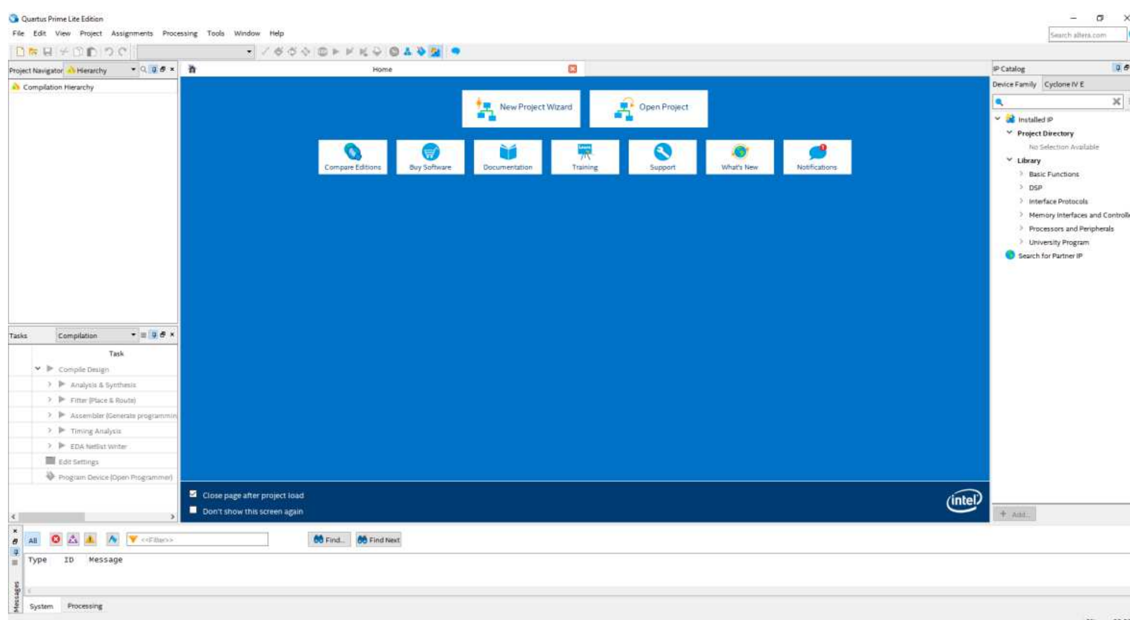


Obrázek 3.8: Zapojení modulu HX711 k řídící jednotce a vážícímu senzoru [18]

3.3 Použitý software

3.3.1 Vývojové prostředí

K programování hradlových polí od společnost *Altera / Intel* slouží vývojové prostředí *Quartus Prime*, které nahradilo dříve používané *Quartus II*. Základní verze označovaná *Lite Edition* je po registraci dostupná zdarma na stránkách společnosti *Intel* [17]. Jednotlivé verze se od sebe liší podporovanými zařízeními a některými pokročilejšími funkcemi. Jednoduše řečeno, čím složitější návrh a dražší zařízení, tím dražší edice je potřeba pro implementaci. Většina běžně dostupných zařízení, včetně desky použité ke zpracování následujících úloh, patři do skupiny Lite a postačí tedy základní verze.



Obrázek 3.9: Vývojové prostředí po prvním zapnutí

Uprostřed se nachází pracovní plocha, ve které se zobrazují soubory otevřené v projektu, a to jak návrhy ve formě kódu, tak i schematické návrhy. Po sestavení projektu se zde objeví i záložka s výsledkem kompilace, zobrazující využití logických bloků, pamětí, pinů atd.

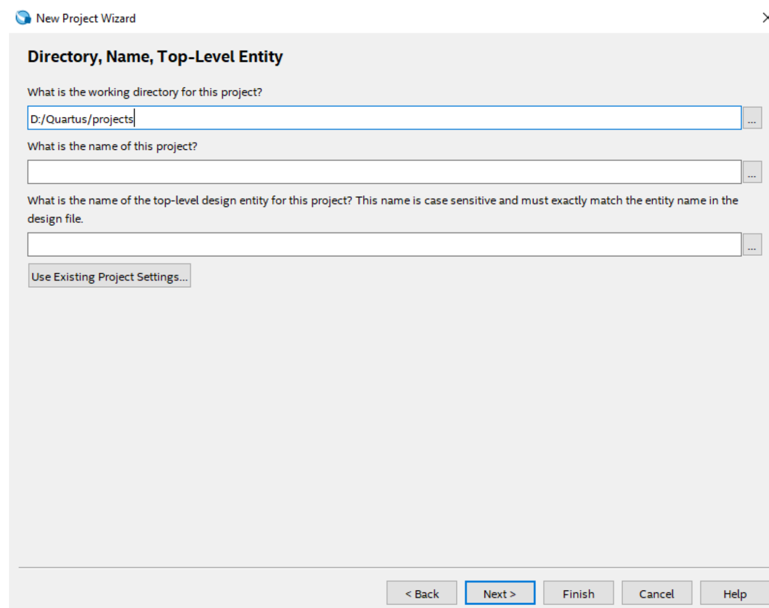
V levé horní části se nachází *Project Navigator*, v kterém se zobrazují vytvořené struktury a návrhy. Záložka *Hierarchy* zobrazuje aktuální strukturu projektu definovanou nadřazeným návrhem. Záložka *Files* obsahuje všechny soubory v projektu. Okno v levé dolní části zobrazuje procesy nutné ke kompilaci projektu. Na konci seznamu se nachází programátor, který slouží k nahrání návrhu do hradlového pole. Ve spodní části se zobrazují zprávy o průběhu kompilace, chyby a varování.

3.3.2 Vytvoření projektu

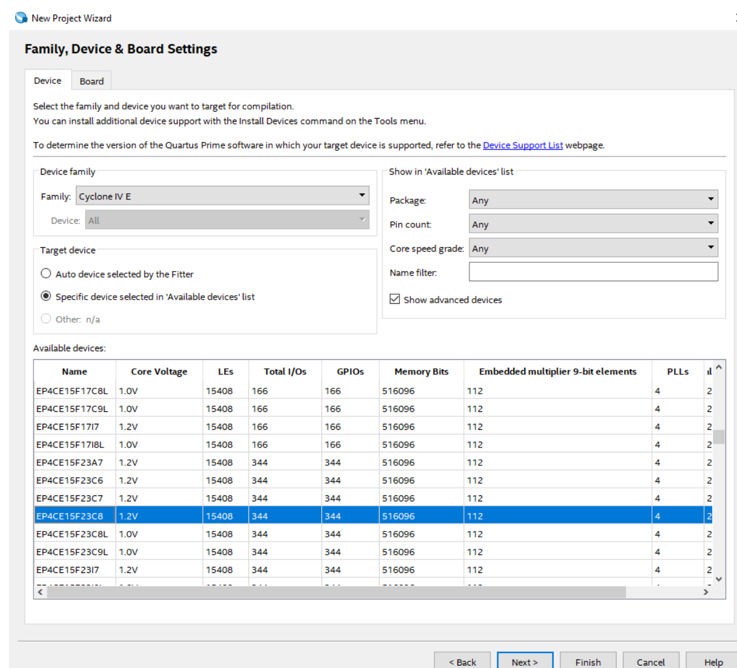
Během vytváření nového projektu je nutné zvolit umístění projektu, název a výchozí nadřazený návrh (angl. *top-level design*). Dále se vybírá typ projektu a je zvolen prázdný (angl. *Empty project*). Další nastavení umožňuje do projektu přidat některé již vytvořené soubory, nechá se prázdné.

Důležitým krokem během nastavování nového projektu je výběr čipu, pro který bude návrh vytvořen. Podle stažených a nainstalovaných balíčků je zobrazen výběr podporovaných hradlových polí. Pro zpracování následujících úloh je vybrána rodina Cyclon IV E a konkrétně čip EP4CE15F23C8.

Následuje nastavení EDA tools (Electronic Design Automation), které se nechá prázdné, a nakonec už je jen shrnutí projektu. Tímto je vytvořen nový projekt připravený k práci.



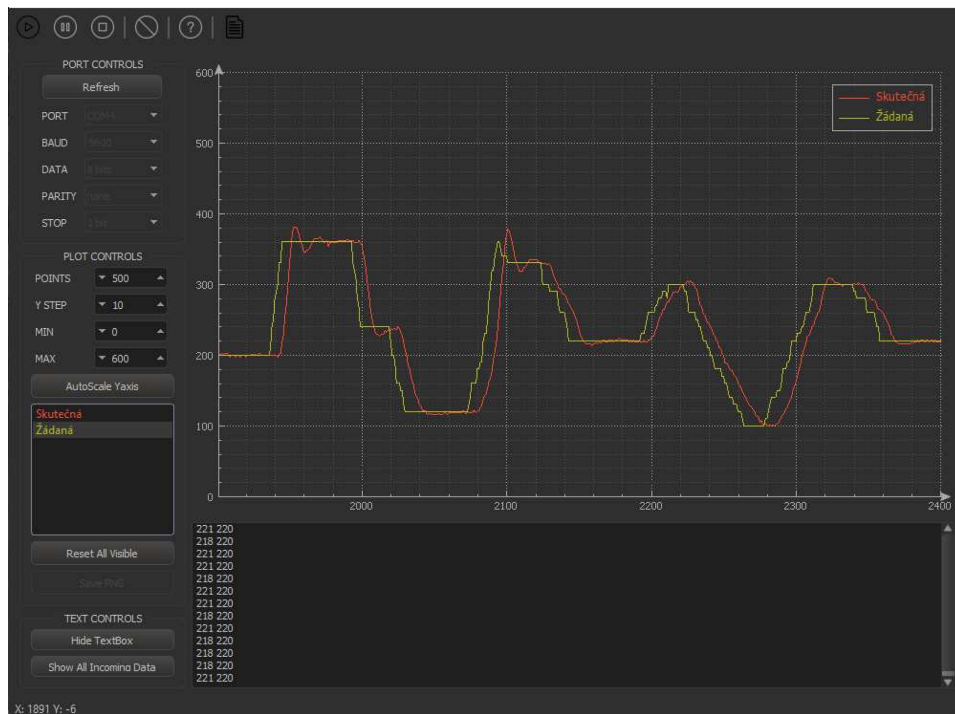
Obrázek 3.10: Založení nového projektu



Obrázek 3.11: Výběr zařízení při vytváření projektu

3.3.3 Sériový plotter

Kromě samotného vývojového prostředí je během úloh využit program [16], který v reálném čase vykresluje přijatá data do grafu. Není potřeba žádná konkrétní přenosová rychlost, umožňuje základní možnosti nastavení jako většina terminálů sloužících k UART komunikaci. Výhodou tohoto programu je, že dokáže vykreslovat několik hodnot zároveň v různých barvách. Rozsah grafu se dá nastavit automaticky dle vykreslovaných hodnot nebo ručně zadáním krajních hodnot. Také disponuje možností ukládat data ve formátech CSV a PNG. Pro správné fungování musí být data přijata v konkrétním formátu. Každý přijatý řetězec musí začínat symbolem dolaru a končit středníkem. Hodnoty jednotlivých kanálů jsou odděleny mezerou. Celý řetězec pak vypadá „$*[číslo1] [číslo2] … [čísloN]*;“. Konkrétní řetězec s dvěma hodnotami by mohl vypadat například takto: „$-45 127;“.



Obrázek 3.12: Rozhraní sériového plotteru

3.4 Návrh sériové komunikace s PC

Jako první byla vytvořena sériová komunikace, aby bylo možné s hradlový polem komunikovat přes sériový terminál počítače. Návrh komunikace UART je složen z přijímače a vysílače. Protože je rychlost přenosu dat sériovou komunikací relativně malá oproti interní rychlosti, kterou budou posléze komunikovat jednotlivé části návrhu, je nutné přidat blok, který tyto rychlosti bude vyrovnávat, aby nedocházelo ke ztrátě dat.

3.4.1 Vyrovnávací paměť

K vyrovnání rychlostí mezi sériovou komunikací a interní komunikací hradlového pole je použita jednoduchá vyrovnávací paměť typu FIFO *(First In First Out*). Jak již název napovídá, data jsou z paměti čtena v pořadí, v kterém se do paměti zapsala. Návrh takové vyrovnávací paměti spočívá ve vytvoření bitového pole o velikosti MxN, kde M udává hloubku pole, tj. kolik bytů je možné do něj uložit, a N je velikost bytů, v tomto případě 8. Dalším prvkem jsou ukazatele pro zápis *Head* a čtení *Tail*. Ukazatele po spuštění či restartu odkazují na první pozici pole. Při zápisu jsou data uložena na aktuální pozici ukazatele zápisu a poté se ukazatel zvýší o jedna. Analogicky funguje ukazatel čtení. Zvláštní situací je stav, kdy oba ukazatele odkazují na stejnou pozici. V tom případě je paměť buďto zcela prázdná, nebo plná. Který z těchto stavů nastal se pozná podle toho, zda došlo k „uzavření smyčky“, tj. ukazatel zápisu prošel přes celé pole a dostal se až na pozici čtení, za kterou jít nemůže, protože by došlo k přepsání dat.

Budeme například uvažovat paměť o velikosti 8 bytů. Pokud se do paměti zapíše 7 bytů a žádný se nepřečte, bude ukazatel čtení odkazovat na pozici 0 a ukazatel zápisu na pozici 7 (indexujeme od nuly). Pokud se zapíše další byte, ukazatel zápisu přeskočí na pozici 0. V tuto chvíli oba ukazatel odkazují na pozici 0 a vyrovnávací paměť je plná. Ukazatel zápisu však uzavřel smyčku přechodem z pozice 7 na pozici 0. Pokud se nyní z paměti přečte všech 8 bytů, ukazatel čtení projde přes všechny pozice a vrátí se zpět na pozici 0. Nyní opět oba ukazatele odkazují na pozici 0, ale paměť je prázdná. Tentokrát přechodem ukazatele čtení z pozice 7 na pozici 0 došlo k rozpojení smyčky.

Z příkladu je jasně vidět, jak bude probíhat algoritmizace. Pokud oba ukazatele odkazují na stejnou pozici a došlo k zavření smyčky, je paměť plná, pokud nedošlo k uzavření smyčky, je paměť prázdná. K uzavření smyčky dochází, když ukazatel zápisu přeskočí z poslední pozice na první, a k rozpojení dochází, když analogicky přeskočí ukazatel čtení.

Následující úlohy jsou psány v jazyce VHDL, proto se při vytváření nového souboru musí zvolit typ *VHDL File*. Název smí být tvořen alfanumerickými znaky bez diakritiky a podtržítkem, stejně tomu je i u názvu projektu [20]. Na začátku souboru je potřeba uvést, že se bude používat knihovna jazyka VHDL a příslušné moduly. Tento začátek bude u všech vytvořených VHDL souborů stejný, někdy přibyde i modul *IEEE.STD_LOGIC_ARITH.ALL*, který umožňuje další matematické operace.

Následuje část entity, kde je uveden název bloku, který musí odpovídat jménu souboru. Entity popisuje pouze vstupy a výstupy z bloku, tzv. porty.

Vstupy jsou hodinový signál *clk*, reset *rst*, zápis *DW*, čtení *DR* a všechny jsou typu *std_logic*. Vstupní data *DIN* jsou typu *std_logic_vector* s indexy zleva sestupně 7 až 0. Stejného typu a velikosti jsou výstupní data *DOUT*. Dalšími výstupy jsou indikátory prázdné a plné paměti, které jsou typu *std_logic*. Krom klasických vstupů a výstupů použijeme parametr *FIFO_DEPTH*, který je z hlediska vnitřní struktury bloku konstantní, proto je typu *constant*, ale je možné jej z vnějšku měnit. Díky tomu je možné měnit hloubku paměti z nadřazeného návrhu, aniž by bylo nutné zasahovat do vnitřního kódu bloku.

```
entity buffer_fifo is
	generic(
		constant FIFO_DEPTH : positive := 8
	);
	port(
		clk : in std_logic;
		rst : in std_logic;
		DW : in std_logic;
		DR : in std_logic;
		DIN : in std_logic_vector(7 downto 0);
		DOUT: out std_logic_vector(7 downto 0);
		Empty : out std_logic;
		Full : out std_logic
	);
end entity;
```

K popsání chování bloku slouží část *architecture*. Název architektury je libovolný, ale řídí se stejnými pravidly, jako název souboru. Jméno přidružené entity se však musí shodovat. Jeden VHDL soubor může obsahovat i více architektur, ale entitu smí mít pouze jednu.

Dalším základním kamenem návrhu je *process*. Jeho použití je sice nepovinné, ale umožňuje vytvářet části kódu, které se vykonávají sekvenčně, stejně jako u vyšších programovacích jazyků. Proces se spouští se změnou některého ze signálů, uvedených jako jeho parametry. Seznam těchto parametrů se nazývá citlivostní seznam (angl. *sensitivity list*).

Pro vytvoření pole, představující paměť, je potřeba vytvořit vlastní typ proměnné, která je definována jako pole vektorů. Vnitřní vektory představují jednotlivé byty, mají tedy velikost 8 a indexy jsou sestupně. Vnější pole má velikost určenou parametrem *FIFO_DEPTH* a je indexován vzestupně. Indexy vnějšího pole představují adresy paměti. Tímto je pouze deklarován nový typ proměnné. Aby bylo možné tuto paměť použít, je potřeba vytvořit novou instanci. Během syntézy je rozpoznána struktura paměti a při implementaci je využito integrovaných paměťových bloků typu RAM.

```
process(clk)
	-- deklarace paměti
	type MEMORY is array (0 to FIFO_DEPTH - 1) of std_logic_vector(7 downto 0);
```

```
	variable FIFO : MEMORY;

	-- ukazatele
	variable head : natural range 0 to FIFO_DEPTH - 1;
	variable tail : natural range 0 to FIFO_DEPTH - 1;
	variable looped : boolean;
```

Během každé periody hodinového signálu je kontrolováno, jestli se na vstupu pro zápis nebo čtení neobjeví log 1. Pokud je na zapisovacím signálu indikována log 1, jsou na aktuální pozici uložena vstupní data a hodnota ukazatel zápisu je zvýšena o 1. Analogicky jsou při čtení na výstup přiřazena data z aktuální pozice pro čtení a hodnota ukazatele zvýšena.

```
-- zápis  dat
if DW = '1' then
	-- kontrola polohy zápisu
	if looped = false or head /= tail then

		-- zapsání do příslušné adresy
		FIFO(head) := DIN;

		-- úprava ukazatele zápisu
		if head = FIFO_DEPTH - 1 then
			head := 0;

			looped := true;
		else
			head := head + 1;
		end if;
	end if;
end if;
```

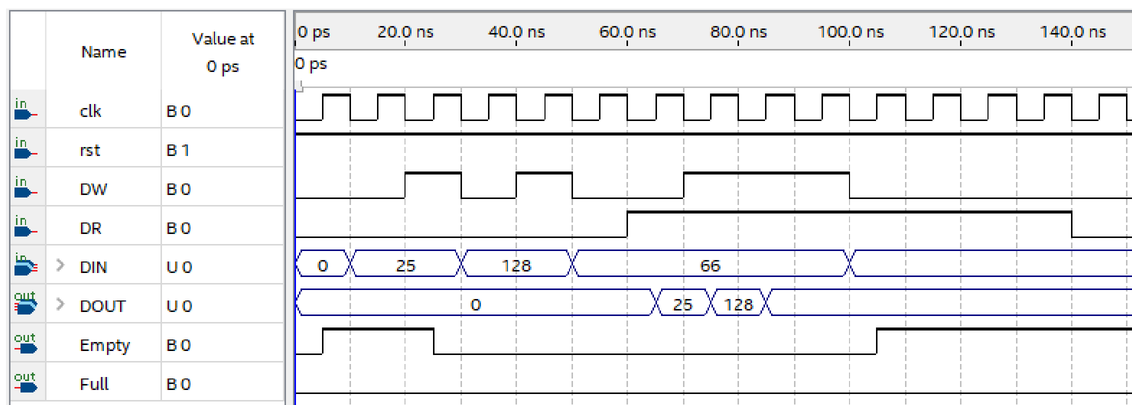
Takto navržená paměť je již plně funkční. Pro lepší komunikaci s ostatními bloky v návrhu je však vhodné přidat indikátory stavu zaplnění paměti.

```
-- úprava indikátorů stavu paměti
if head = tail then
	if looped then
		Full <= '1';
	else
		Empty <= '1';
	end if;
else
	Full <= '0';
	Empty <= '0';
end if;
```

Funkčnost navržené vyrovnávací paměti se otestuje na jednoduchém příkladu. Na vstupní datovou sběrnici jsou připojeny spínače. Na výstupní datovou sběrnici je připojeno 8

světelných diod. Další dvě světelné diody se mohou připojit na výstupy signalizace prázdné a plné paměti. V neposlední řadě jsou připojena tlačítka na vstupy zápisu a čtení. Protože jsou tlačítka mechanického charakteru, je třeba je ošetřit, aby se paměť chovala, jak očekáváme. Impuls zápisu a čtení musí být o délce jedné periody hodinového signálu, jinak se stane, že budou data zapsána či přečtena několikrát za sebou. Proto je mezi tlačítkem a vyrovnávací pamětí blok, který přesně tohle zajišťuje. Pro ideální chování je vhodné během jednoho cyklu vyslat impuls k přiřazení dat na výstup a během následujícího cyklu tato data přečíst. Je tím zaručeno, že se stará data stihla řádně přepsat novými. Stejně tak pro zápis, během jednoho cyklu přiřadit na vstup nová data a další cyklus je zapsat do paměti.

Před samotným nahráním programu do hradlového pole je vhodné nejdříve spustit návrh v simulaci. Je potřeba vytvořit nový soubor typu *University Program VWF*. Po jeho otevření se objeví editor s časovou osou pro pár stovek nanosekund. Program dovolí maximální dobu simulace 10 µs. Nejrychlejší způsob přidání vstupů a výstupů je pomocí *Edit->Insert->Insert Node or Bus->Node Finder…->List*. Zobrazí se seznam dostupných vstupů a výstupů, které už je možné libovolně přidat do editoru. Aby se vstupy a výstupy v seznamu objevily, je nutné provést kompletní analýzu a syntézu návrhu (angl. *Analysis & Synthesis*). Díky simulaci je možné detailně a jednoduše prozkoumat chování návrhu ještě před nahráním do hradlového pole a snáze tak odhalit chyby v návrhu. Také se dá použít ke kontrole chování během některých krajních situací, které by bylo obtížné otestovat experimentálně.



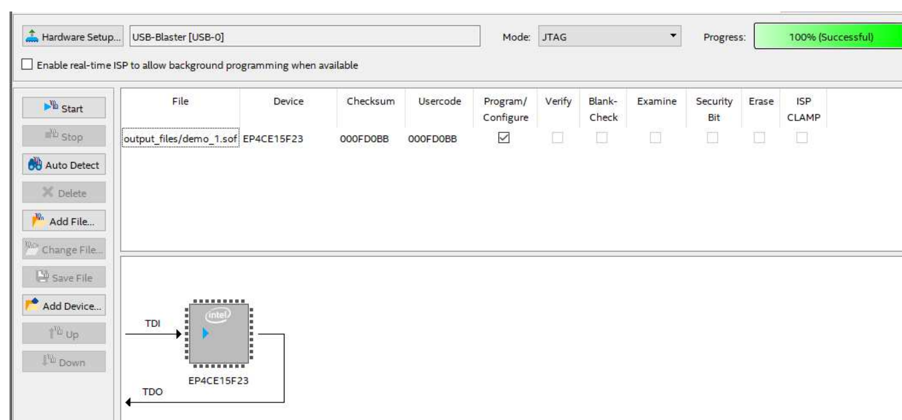
Obrázek 3.13: Simulace vyrovnávací paměti

Umístění vstupních a výstupních pinů je u takto jednoduchého návrhu zcela libovolné, některé piny však nemusí požadovaný směr toku dat podporovat. Zvolená deska má ale všechny vytažené piny typu IO. Pokud by se jednalo o složitější návrh, kdy by se celkové využití logických bloků blížilo maximální kapacitě, bylo by potřeba vhodně zvolit lokaci pinů, neboť v opačném případě by se nemuselo podařit návrh sestavit.

U všech návrhů je hlavní hodinový signál připojen na pin T2, který je připojený na externí krystal s frekvencí 50 MHz, a reset je připojen na pin W13, ke kterému je připojeno tlačítko SW1 na desce. Klidový stav tlačítka je log 1, proto jsou resetovací části kódu citlivé na log 0. K přiřazení pinů k signálům slouží grafické prostředí *Pin Planner*. Aby

bylo možné piny přiřadit, musí se úspěšně provést aspoň *Analysis & Elaboration* část kompilace.

Jak již bylo uvedeno při popisu vývojového prostředí, naprogramování hradlového pole se provádí pomocí programátoru (angl. *Programmer*), který se nachází na konci seznamu v okně *Tasks* v záložce *Compilation*. Před samotným nahráním programu do zařízení je nutné provést kompletní kompilaci, včetně části *EDA Netlist Writer*. Poté se v okně programátoru vybere soubor a zařízení, což se většinou provádí automaticky, a v *Hardware Setup* je potřeba zvolit programovací hardware, v tomto případě je to *USB-Blaster*. Pokud je všechno správně nastaveno, zpřístupní se tlačítko *Start*. Po jeho stisknutí dojde k naprogramování hradlového pole. Pokud byl program úspěšně nahrán, v pravé horní části se objeví zelený nápis *Successful*.



Obrázek 3.14: Rozhraní programátoru

3.4.2 Vysílač

Aby mohlo docházet ke korektnímu odesílání dat, je k samotnému bloku vysílače připojena vyrovnávací paměť. Pokud je odesílací buffer prázdný, tzn. nejsou žádná data k odeslání, přepne se do stavu IDLE a bude vyčkávat, než budou nějaká data dostupná. Hned jak se objeví v bufferu nějaká data, zahájí vysílač proces odesílání přiřazením START bitu na sériovou komunikaci a požadavkem na nová data z bufferu, která o chvíli později přečte. Po uplynutí doby pro START bit, začne vysílač na sériovou linku postupně odesílat jednotlivé datové bity a komunikaci zakončí STOP bitem. Během této finální fáze vysílač opět kontroluje, zda jsou k dispozici data k odeslání. Pokud jsou, začne ihned po odeslání STOP bitu, resp. po minimální době určené ke STOP bitu, odesílat další data. Pokud nejsou, uvede se opět do stavu IDLE. Během čtení se data na vstupu zapíšou do vnitřního odesílacího registru.

Vstupy odesílacího bloku jsou hodinový signál *clk*, reset *rst*, indikátor prázdné paměti *Empty* a paralelní datová linka *DATA*. Výstupem je signál čtení dat z paměti *DR* a odchozí linka sériové komunikace *TX*.

Hlavní hodinový signál, v tomto případě signál připojeného krystalu, má několikanásobně vyšší frekvenci, než jaké se používají pro komunikaci UART. Je proto nutné frekvenci snížit. Existuje několik běžně používaných přenosových rychlostí, je proto dobré mít možnost tuhle rychlost měnit. K tomu slouží jistá obdoba předděličky, používané v mikrokontrolerech, která frekvenci sníží na hodnotu, určenou přenosovou rychlostí. Protože se krom odesílání dat na sériovou linku odehrávají i jiné rychlejší procesy, nemůže být zpomalen celý blok, ale musí se zpomalit až jednotlivé části.

$$DELAY = \frac{CLK_FREQ}{BAUD} - 1 \qquad (3.1)$$

Proces odesílání funguje na způsob stavového automatu, který se skládá se stavů čekání na přenos, zahájení přenosu, odesílání dat a ukončení přenosu. Výchozím stavem je stav čekání na přenos, označený jako stav 0.

Signály a proměnné fungují velmi podobně s jedním důležitým rozdílem. Procesní proměnné mění svou hodnotu okamžitě během procesu (obdobně jako proměnné u vyšších programovacích jazyků), zatímco signály se mění až po ukončení procesu.

Většina mikrokontrolerů má k sobě připojené tlačítko, které v případě potřeby celý běžící program resetuje. Hradlová pole však tuhle funkci nemají, proto je dobré si ji přidat. Na začátku procesu všech vytvořených bloků je proto podmínka kontrolující resetovací tlačítko. V případě aktivace resetu se nevykonává hlavní část kódu a všechny signály a proměnné jsou uvedeny do výchozích stavů.

Na začátku stavu 0 je aktivní čítač, který se zvyšuje s frekvencí hlavního hodinového signálu. Jakmile tento čítač dosáhne hodnoty *DELAY*, definované přenosovou rychlostí komunikace, zkontroluje se, jestli jsou v předřazené vyrovnávací paměti dostupná nějaká data (signál *Empty*). Pokud má signál *Empty* hodnotu log 0, tedy buffer není prázdný, zvýší se stav o 1, vynuluje se čítač a signálem *DR* se vyšle impuls do bufferu pro čtení dat. Také je signálu *TX_int* přiřazena log 0, tedy START bit. Pokud ale po dosažení hodnoty *DELAY* má signál *Empty* hodnotu log 1, tedy buffer je prázdný, přepne se odesílatel do stavu IDLE, ve kterém vyčkává na dostupnost dat. Protože se do stavu IDLE dostane až po dosažení čítače hodnoty *DELAY*, dá se tento stav použít jako jedna z podmínek pro zahájení přenosu.

Pokud jsou v paměti dostupná data, je na konci stavu 0 do odesílacího buffer vyslán impuls pro čtení dat. Po 10 cyklech hodinového signálu jsou tato data přečtena a uložena do odesílacího registru. Následující část návrhu pracuje už pouze s přenosovou frekvencí, je proto opět využit parametr *DELAY* ke zpomalení hlavních hodin.

```
if state = 0 then
	if IDLE = '0' then
		count := count + 1;
	end if;

	-- kontrola dostupnosti dalšího bytu
```

```
		if (count >= DELAY or IDLE = '1') and Empty = '0' then
			state := 1;
			count := 0;
			IDLE <= '0';
			DR <= '1';
			TX_int <= '0';
			D_read <= '0';
		elsif count >= DELAY and Empty = '1' then		-- pozastavení do dalšího odesílání dat
			IDLE <= '1';
			count := 0;
		else
			DR <= '0';
		end if;
	else
		DR <= '0';

		-- cteni novych dat
		if count > 10 and D_read = '0' then
			DATA_temp <= DATA;
			D_read <= '1';
		end if;

		count := count + 1;

		-- zpomaleni na rychlost prenosu
		if count >= DELAY then

			-- odesílání dat
			if state < 9 then
				TX_int <= DATA_temp(state - 1);
			state := state + 1;

			-- odeslání STOP bitu
			else
				TX_int <= '1';
				state := 0;
			end if;

			count := 0;
		end if;
	end if;
```

Při každém průchodu je na interní signál *TX_int* přiřazen odpovídající bit z odesílacího registru. Index bitu odpovídá hodnotě stavu zmenšené o 1. Následně je zvýšeno číslo stavu odesílání. Tímto způsobem jsou postupně na signál přiřazeny všechny bity z odesílacího registru, až se dojde do stavu 9, kdy je na výstup přiřazen STOP bit a stav nastaven na nulu. Mimo proces je interní signál *TX_int* přiřazen na odchozí signál *TX*, tato operace probíhá nezávisle na procesu.

3.4.3 Přijímač

Po spuštění je přijímač ve stavu IDLE a čeká na začátek komunikace (START bit, log 0). Jakmile se detekuje začátek komunikace, počká půlku periody jednoho bitu a poté ověří, zda je na lince stále START bit. Pokud tam je, komunikace pokračuje, pokud není, komunikace se přeruší – START bit vyhodnocen jako falešný (způsobený například

rušením) a přijímač se vrátí do stavu IDLE. Pokud komunikace pokračuje, počká přijímač další periodu a poté přečte data. Zpoždění o půl periody během START bitu způsobí, že se během čtení nachází zhruba v polovině doby odesílání datového bitu. Díky tomu není přijímač citlivý na odchylky v rychlostech přijímače a odesílatele, protože na obě strany má půl periody rezervu. Takto projde všech 8 bitů, až se dostane ke STOP bitu, který je v tomto případě jeden. Zde opět probíhá kontrola. Pokud se přečtená hodnota neshoduje s hodnotou STOP bitu, jsou všechna data zahozena a přijímač se vrátí do stavu IDLE. Pokud se hodnota shoduje, je komunikace vyhodnocena jako úspěšná, přenesená data se odešlou dál, většinou do vyrovnávací paměti, a přijímač je připraven na příjem dalšího bytu.

Vstupy přijímacího bloku jsou hodinový signál *clk*, reset *rst* a příchozí linka sériové komunikace *RX*. Výstupy jsou signál zápisu dat do paměti *DW* a paralelní datová linka *DATA*. Mezi vstupy by se mohl přidat i indikátor zaplnění paměti, nicméně pro vhodnou hloubku paměti a mnohem vyšší rychlost interní komunikace, než je rychlost sériové komunikace, není pro nadcházejí příklady potřeba přeplnění ošetřovat.

Stejně jako odesílatel pracuje i přijímač ve stylu stavového automatu. A opět nutné zpomalit hodinový signál stejným parametrem *DELAY*.

Na rozdíl od odesílatele, je výchozí hodnotou signálu *IDLE* log 1. V tomto stavu přijímač setrvá, dokud se hodnota vstupní sériové linky nezmění na log 0, tím se zahájí komunikace. Změnou hodnoty *IDLE* na log 0 je zapnut čítač, který se bude zvyšovat až do poloviny hodnoty *DELAY*. Pokud přenos dat skutečně běží, musí být stále na vstupu log 0, a pokud je nastavena správná přenosová rychlost, proběhne tohle čtení přibližně v polovině přenosu bitu. Pokud je přečtena log 0, stav se o 1 zvýší, pokud ne, signál *IDLE* je přepnut na výchozí hodnotu.

Protože je synchronizace komunikace provedena během stavu 0, může být frekvence vykonávání zbylých stavů už zpomalena na rychlost komunikace.

Během následujících stavů 1 až 8 se zapisují hodnoty ze vstupním signálu *RX* do přijímacího registru. Datové bity přichází ve stejném pořadí, v jakém se zvyšuje stav. Index bitu odpovídá číslu stavu zmenšeném o 1. Ve stavu 9 se kontroluje přítomnost STOP bitu. Pokud je na vstupu log 1, přiřadí se data z přijímacího registru na datový výstup a signál *received* se změní na log 1. Nezávisle na hodnotě STOP bitu je stav změněn na 0 a signál IDLE na log 1.

```
-- cekani za zacatek prenosu
if IDLE = '1' then
	if RX = '0' then
		IDLE <= '0';
	end if;
else
	count := count + 1;

	-- potvrzeni zacatku prenosu
	if state = 0 then
```

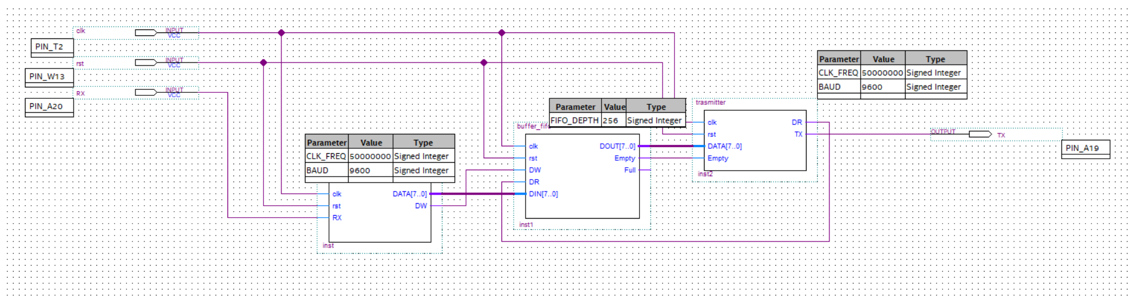
```
            if count >= DELAY / 2 then
                if RX = '0' then                        -- opetovna detekce START bitu
                    state := state + 1;
                else                                    -- zjisteni falesneho zacatku
                    IDLE <= '1';
                end if;
                count := 0;
            end if;

        -- prechod k dalsimu bitu
        else
            if count >= DELAY then

                -- kontrola STOP bitu
                if state = 9 then
                    if RX = '1' then
                        DATA <= DATA_temp;
                        received <= '1';
                    end if;
                    IDLE <= '1';
                    state := 0;

                -- ulozeni prichoziho bitu
                else
                    DATA_temp(state - 1) <= RX;
                    state := state + 1;
                end if;
                count := 0;
            end if;
        end if;
    end if;
```

Přijatá data sice byla přiřazena na výstup, ale ještě je potřeba je uložit do připojené vyrovnávací paměti. Pokud je signál *received* nastaven na log 1, vytvoří se impuls pro zápis, který bude oproti změně dat na výstupu zpožděný, jak bylo vysvětleno u návrhu vyrovnávací paměti. Také je nutné, aby tento impuls měl délku jedné periody hlavního hodinového signálu, jinak se data do paměti zapíšou několikrát. Proto tato část nesmí být v oblasti návrhu zpomalené na přenosovou rychlost.

3.4.4 Ukázkový projekt Echo

K propojení více bloků je po přehlednost využito blokového diagramu. Za tímto účelem je vytvořen nový soubor typu *Block Diagram/Schematic File*. Vytvořené bloky kódu je potřeba převést na symboly. To se provede kliknutím pravým tlačítkem na příslušné soubory v *Project Navigator*. Obdobně se nově vytvořený schematický soubor musí nastavit jako nadřazený návrh *Top-level entity*.

Ukázkou funkčnost navržené sériové komunikace je jednoduchý echo návrh, který do počítače pošle zpět to, co je z něj odesláno do hradlového pole. Odeslaná data se ze sériové sběrnice rozloží na paralelní sběrnici a pak zpátky na sériovou. Zapojení je tedy následující – data přicházejí po signálu *RX* do přijímače, z něj vychází po paralelní sběrnici do vyrovnávací paměti, kde jsou uložena, dokud je z ní nepřečte odesílatel, který je po signálu *TX* odešle zpět.



Obrázek 3.15: Schématické zapojení nadřazeného návrhu Echo

3.5 Generátor PWM

Nejdříve je potřeba vytvořit jednoduchý čítač, který se během každé periody hodinového signálu zvýší o 1. Kdyby byl použit hlavní hodinový signál přímo k řízení čítače, byla by výsledkem pro osmibitový čítač velmi vysoká frekvence, která by byla pevně daná. K nižší a variabilní frekvenci se dojde, obdobně jako tomu bylo u sériové komunikace, zpomalením hlavního hodinového signálu následujícím vztahem (3.2).

$$DELAY = \frac{CLK_FREQ}{256 * PWM_FREQ} \tag{3.2}$$

Hlavní hodinový signál se zpomalí na rychlost, která ve výsledku bude generovat signál o zadané frekvenci. Tento způsob je ale vhodný pro frekvence přibližně do 15 kHz, pro vyšší frekvence pak dochází ke znatelné odchylce způsobené zaokrouhlováním.

Tímto je vytvořen čítač s volitelnou frekvencí. Ke generování pulsně šířkové modulace je potřeba přidat referenční hodnotu. Jednoduchým porovnáváním hodnoty čítače a referenční hodnoty se získá výsledná modulace. Během každé změny čítače se výstupní hodnota aktualizuje.

Vstupy modulu PWM jsou hodinový signál *clk*, reset *rst* a hodnota PWM *pwm_value*. Výstupem je jediný signál *PWM* typu *std_logic*. Parametr *PWM_FREQ* slouží k nastavení frekvence modulace a je zadáván v hertzech.

```
-- zpomalení hodinového signálu
if cnt_clk >= DELAY then
	cnt_clk := 0;

	-- krajní hodnota
	if pwm_value = 0 then
		PWM <= '0';

	else
		-- porovnání hodnot
		if counter <= pwm_value then
```

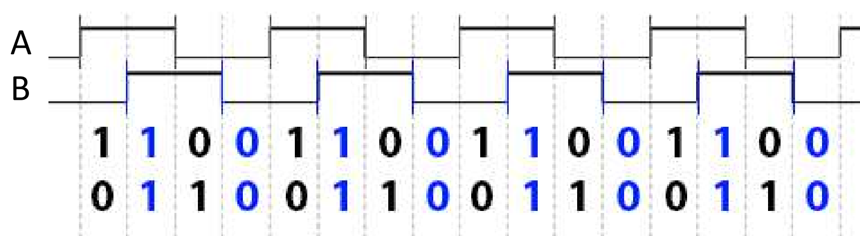
```
				PWM <= '1';
			else
				PWM <= '0';
			end if;
		end if;

		counter <= counter + 1;
	end if;
```

Aby při zadání nulové hodnoty byl výsledkem klidový signál log 0, je přidána ošetřující podmínka.

Jednoduchou demonstrací je ovládání jasu světelné diody. Jednou z možností je odebrat *pwm_value* ze vstupů a vytvořit stejnojmenný interní signál s výchozí hodnotou. Druhou možností je připojit generátor PWM přímo na UART přijímač, bez použití vyrovnávací paměti, a ovládat jas pomocí ASCII znaků, resp. jejich osmibitových hodnot.

3.6 Enkodér

Stejně jako tlačítko je i enkodér mechanického charakteru, a proto je potřeba jej ošetřit stejným způsobem. Nejjednodušší způsob je zpomalit hodinový signál na rychlost, která bude dost nízká na to, aby ignorovala vysokofrekvenční „záškuby“ signálu, ale dost vysoká na to, aby nedocházelo k přeskakování skutečných změn signálu.



Obrázek 3.16: Příklad signálů generovaných enkodérem

Pokud je indikována změna na *kanálu A* a hodnoty *kanálu A* a *kanálu B* jsou různé, nebo pokud je indikována změna na *kanálu B* a hodnoty *kanálu A* a *kanálu B* jsou stejné, pak se enkodér otáčí ve směru hodinových ručiček. Pro opačný směr je tomu právě naopak. Pro kvadraturní enkodér jsou tyto změny čtyři během jedné periody jednoho ze signálů, tzn. hodnota může vrůst nebo klesnout až o 4 (viz Obrázek 3.16).

Signály vstupující do bloku jsou hodinový signál *clk*, reset *rst*, kanál A *encA* a kanál B *encB*. Výstupem je znaménkové číslo s parametrickým rozsahem *enc_value*. Parametry bloku jsou *DELAY*, který slouží ke zpomalení hodinového signálu, a *OUTPUT_WIDTH*, který určuje bitový rozsah enkodéru.

Vnitřní pomocné signály jsou *count*, který uchovává vnitřní hodnotu enkodéru, *old_encA* a *old_encB*, které slouží k uložení předchozích hodnot kanálů A a B. Těmto signálům je na začátku přiřazena aktuální hodnota kanálů.

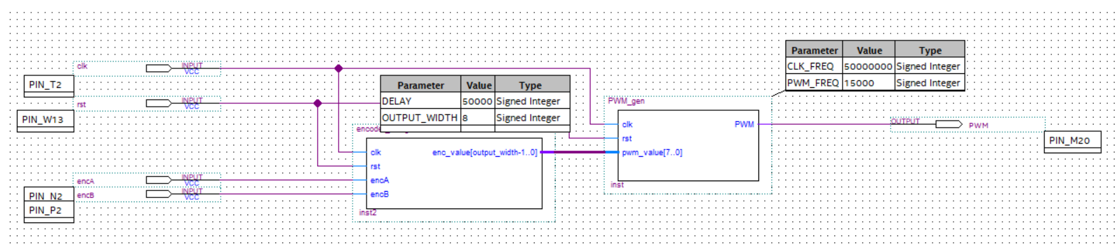
Stejně jako tomu bylo u přechozích příkladů, proces je citlivý na hlavní hodinový signál a spouští se na jeho náběžnou hranu. Následuje kontrola resetu a poté zpomalení rychlosti.

```
if cnt >= DELAY then
	cnt := 0;
	if encA = not old_encA then
		if encA = encB then
			if count > -2**(OUTPUT_WIDTH-1) then
				count <= count - 1;
			end if;
		else
			if count < 2**(OUTPUT_WIDTH-1)-1 then
				count <= count + 1;
			end if;
		end if;

		old_encA <= encA;
	elsif encB = not old_encB then
		if encA = encB then
			if count < 2**(OUTPUT_WIDTH-1)-1 then
				count <= count + 1;
			end if;
		else
			if count > -2**(OUTPUT_WIDTH-1) then
				count <= count - 1;
			end if;
		end if;
		old_encB <= encB;
	end if;
end if;
```

Kontroluje se, jestli se změnila hodnota kanálu A oproti předchozí uložené. Dojde-li ke změně, zjišťuje se, jestli je hodnota kanálu A stejná jako hodnota kanálu B. Podle toho se určí směr otáček. Aby nedošlo k přetečení, jsou krajní hodnoty ošetřeny. Maximální hodnoty jsou zadané parametricky, podle rozměru enkodéru. Nakonec je uložen současný stav kanálu A. Obdobně je vyhodnocována změna na kanálu B.

Mimo proces je signál *count*, který obsahuje hodnotu enkodéru, přiřazen na výstup *enc_value*.

Příhodnou ukázkou funkce enkodéru je nastavení jasu světelné diody pomocí PWM, které bylo vytvořeno v předchozí úloze. Enkodér je jen potřeba upravit tak, aby jeho rozsah byl 8 bitů, výstupní hodnota byla typu *unsigned* a krajní hodnoty byly 0 a 255. Výstup enkodéru *enc_value* se připojí na vstup *pwm_value*. Otáčením enkodéru se pak plynule mění jas připojené světelné diody. Výsledné schéma je tvořeno jen blokem enkodéru a generátorem PWM.



Obrázek 3.17: Schématické zapojení řízení jasu LED pomocí enkodéru

3.7 Regulace DC motoru

Řízení otáček stejnosměrného motoru se ve své podstatě nijak neliší od ovládání jasu světelné diodu v předchozí úloze. Rozdílem je ale hardwarové zapojení. Motor vyžaduje vlastní zdroj s vyšším napětím a připojení na H-můstek L298.

Modul L298 pracuje na 5 V logice a hradlové pole na 3.3 V logice, proto je nutné použít napěťový převodník. Na vstup *ENA* modulu je přiveden signál PWM z hradlového pole, na vstup *IN1* je připojena log 1 a na *IN2* je přivedena log 0. Signál *ENA* zapíná můstek a signály *IN1* a *IN2* spínají tranzistory. Tohle zapojení umožňuje pouze jeden směr otáčení.

Jmenovité napětí použitého motoru je 12 V. Použitý H-můstek je složen z bipolárních tranzistorů, tudíž je na každém úbytek napětí kolem 0.7 V. Vhodné napětí zdroje k napájení modulu by se pak mělo pohybovat kolem 13.5 V, aby bylo zaručeno jmenovité napětí na motoru.

3.7.1 Návrh regulátoru

Při návrhu PID regulátoru na hradlovém poli by se mělo mluvit spíše o PSD (proporcionálně sumačně diferenční), neboť regulace není spojitá. Nicméně integrace lepé vystihuje chování složky, a proto se i v diskrétní formě často nazývá PID.

Aby bylo možné regulovat rychlost otáčení motoru, je potřeba znát skutečnou rychlost. K tomu je využit enkodér připojený k hřídeli motoru. Jednoduchou úpravou se ze snímání polohy, kterou se zabývaly předchozí úlohy, získá snímání rychlosti. Návrh enkodéru je upraven tak, že hodnota čítače není vyvedena přímo na výstup, ale výstup je aktualizovaných v pravidelných intervalech a poté je čítač vynulován. Bylo by dobré, kdyby tyto intervaly byly co nejkratší, ale protože enkodér na motoru má nízké rozlišení, byl by pro nízké otáčky nepoužitelný, a i při vyšších otáčkách by docházelo ke značnému zkreslení. Experimentálně bylo zjištěno, že 20 snímků za sekundu dává přijatelné výsledky.

Výstupem enkodéru je sice rychlost otáčení, ale je uvedena v počtu tiků za sekundu. Bylo by výhodnější pracovat s číslem v otáčkách za minutu, proto je přidán jednoduchý převod dle vztahu 3.3, kde N je počet tiků a n jsou otáčky za minutu. Číslo 444 udává experimentálně určený počet tiků na otáčku.

$$n = \frac{N \cdot 60}{444} \tag{3.3}$$

Snímkovací frekvenci rychlostního enkodéru určuje generátor impulsů. Generátor impulsů s délkou jedné periody hlavního hodinového signálu je ve své podstatě jenom čítač, který vytvoří impuls, když dosáhne své maximální nastavené hodnoty. Tato hodnota je určena frekvencí hodinového signálu a snímkovací frekvencí. Jeho vstupy jsou pouze hodinový signál *clk*, reset *rst* a výstupem je požadovaný signál *trig*.

```
if count < period then
        trig <= '0';
        count := count + 1;
else
        trig <= '1';
        count := 0;
end if;
```

Vstupy regulátoru jsou, krom klasických hodin a resetu, znaménkové signály požadované hodnoty *target* a aktuální hodnoty *actual*. Výstupem je akční veličina AV a směr *DIR*. Tyto dva signály by mohly být spojeny do jednoho, ale vzhledem k povaze řízení H-můstku byly rozděleny. Návrh regulátoru má však několik parametrů, které slouží k nastavení regulačních konstant. Ty jsou definovány ve tvaru zlomků, díky tomu není potřeba použít desetinná čísla. Posledním parametrem je frekvence regulace.

Proces regulátoru se opět spouští s náběžnou hranou hodinového signálu a obsahuje resetovací část, která nastavuje výchozí hodnoty. Následuje podmínka, která vyčkává na impuls snímkovacího signálu. Až se zaznamená snímkovací impuls, spustí se výpočet akční veličiny. Hlavní část návrhu je tvořena cyklem *switch* (ve VHDL značený pouze *case*), který během každého spuštění procesu provede jednu operaci z výpočtu akční veličiny. Během prvního průchodu se vypočítá odchylka z hodnot *target* a *actual* a uloží se do signálu *error*. V následujícím průchodu se spočítá sumace odchylek a diference od předchozí odchylky. Dále se vypočítají zásahy od jednotlivých regulačních složek. Poté se jednotlivé zásahy sečtou a uloží do signálu *PID_sum* a v dalším kroku se absolutní hodnota uloží do signálu *AV_buff*. Následně se provede kontrola, jestli se nepřesáhla maximální hodnota 255, číslo se převede na bitový vektor a přiřadí se na výstupní signál AV. Zároveň se na výstup *DIR* přiřadí znaménko akční veličiny.

```
case N is
        when 0 =>                       -- výpočet ochylky skutečné hodnoty od referenční
                error <= to_integer(target) - to_integer(actual);
        when 1 =>                       -- výpočet diference od předchozí odchylky a sumace odchylek
                error_sum <= error_sum + error;
                error_dif <= error - error_old;
        when 2 =>                       -- výpočet jednotlivých složek regulátoru
                P <= (P_num * error)/ P_den;
                I <= (I_num * error_sum)/(SAMPLES * I_den);
```

```
		D <= (D_num * error_dif * SAMPLES)/D_den;
	when 3 =>			-- sumace složek regulátoru
		PID_sum <= P + I + D;
	when 4 =>			-- absolutní hodnota
		_buff <= abs(PID_sum);
	when 5 =>			-- omezení akční hodnoty, určení směru rotace a přiřazení na výstup
		if AV_buff > 255 then
			AV <= (others => '1');
		else
			AV <= std_logic_vector(to_unsigned(AV_buff, 8));
		end if;

		if PID_sum >= 0 then
			DIR <= '1';
		else
			DIR <= '0';
		end if;
	when 6 =>			-- uložení odchylky pro další cyklus
		error_old <= error;
		eval <= '0';
	when others => NULL;
end case;
```

Akční veličina *AV* pak vstupuje do generátoru PWM a výsledný modulovaný signál je přiveden na pin *ENA* na modulu L298. Směr otáčení reprezentovaný hodnotou signálu *DIR* je přiveden na pin *IN1* a jeho negace na pin *IN2*.

3.7.2 Nastavení regulátoru s vykreslením průběhu

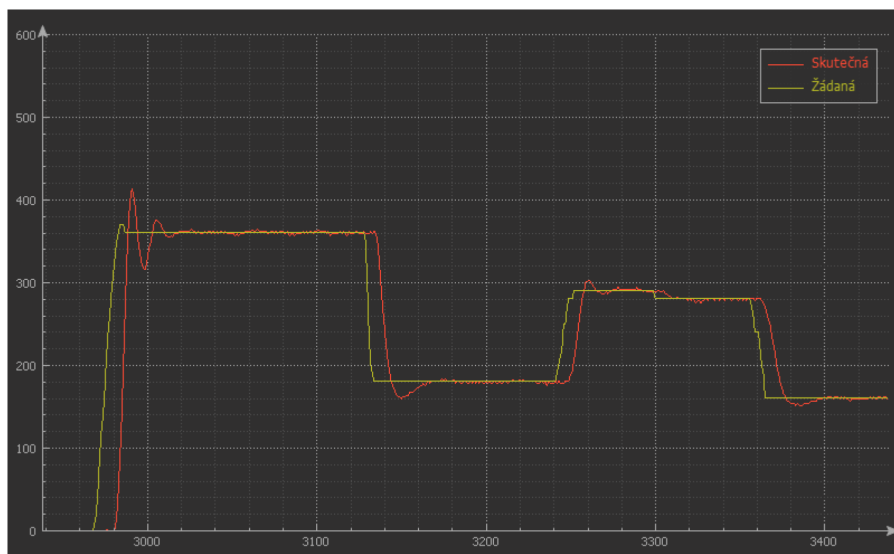
Aby bylo regulátor možné snáze naladit, je do návrhu během ladění připojen blok, který převádí požadovanou hodnotu a aktuální hodnotu na data typu *string*, která jsou poté odeslána přes UART do počítače a vykreslena v sériovém plotteru [16]. Díky tomu je možné v reálném čase vidět průběh regulace a rychleji tak zvolit vhodné regulační konstanty. Tento blok je v návrhu pouze za účelem naladění regulátoru, pro následné řízení motoru není vhodný, a to ze dvou hlavních důvodů. Jedním důvodem je to, že tento převod je velmi náročný na sestavení a zbytečně zahltí velkou část kapacity hradlového pole. Druhým důvodem je zahlcení samotné sériové komunikace, které se projeví obzvlášť pokud je motor řízen pouze mocí sériového terminálu.

Převod čísla by se dal rozdělit na 3 části. Během první se určuje znaménko čísla a případně se pomocí bitové masky číslo převede na absolutní hodnotu. V druhé části probíhá samotný převod z bitového čísla na ASCII symboly přes postupné zmenšování čísla po jednotlivých řádech. Převod obou čísel probíhá nezávisle na sobě. Až jsou obě čísla převedena na symboly, přejde se k poslední části, kde jsou symboly postupně odeslány do odesílacího bufferu ve formě, kterou určuje použitý sériový plotter. Po odeslání celého rámce jsou všechny pomocné proměnné resetovány.

```
-- převod čísla A
if maxValueA > 1 then
	if numberValue_A >= maxValueA then
```

```
                numberA(dig_A) := numberA(dig_A) + 1;
                numberValue_A := numberValue_A - maxValueA;
        end if;
        if numberValue_A < maxValueA then
                if numberA(dig_A) > 0 or notZeroA = '1' then        -- vyloučení nul před číslem
                        numberA(dig_A) := numberA(dig_A) + 48;          -- na ASCII
                        dig_A := dig_A + 1;
                        notZeroA := '1';            -- objevila se nenulová číslice
                end if;

                maxValueA := maxValueA/10;               -- zmenšení šetřeného řádu
        end if;
else
        numberA(dig_A) := std_logic_vector(numberValue_A(7 downto 0)) + 48;
        evalA_done <= '1';                          -- potvrzení ukončení výpočtu
end if;
```

První ukázková úloha slouží k nastavení regulátoru. Otáčky jsou nastaveny pomocí rotačního enkodéru. Hodnoty žádané a skutečné rychlosti otáčení jsou odesílány přes UART do počítače, kde je průběh regulace vykreslen v sériovém plotteru.

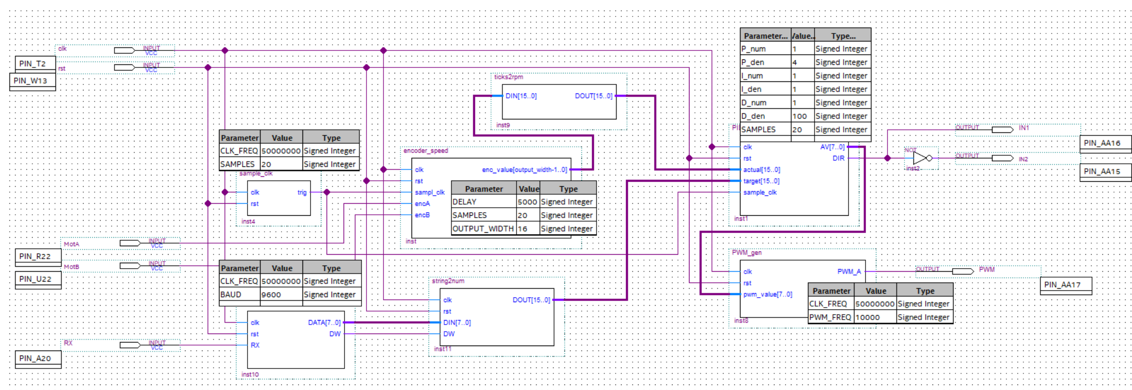


Obrázek 3.18: Průběh regulace otáček motoru

3.7.3 Řízení regulovaného DC motoru přes UART

Druhá ukázková úloha s regulátorem umožňuje ovládat motor přes sériový terminál počítače. Namísto vstupu z enkodéru je připojen UART přijímač, který data neukládá do vyrovnávací paměti, ale do bloku, který převádí ASCII symboly na čísla. Když jsou přijata nová data, zkontroluje se, jestli se jedná o číslici, jinak je znak ignorován. Symbol je převeden na číslici a vyčkává se na další data. Pokud se opět jedná o číslici, předchozí číslo se vynásobí deseti, nový symbol se převede na číslo a obě hodnoty se sečtou. Tento proces se opakuje, dokud není přijat ukončovací znak LF. Konečná hodnota je pak přiřazena na výstup. Převod podporuje i záporné hodnoty, proto se u prvního znaku kontroluje, jestli se nejedná o symbol mínus. K případnému převodu na záporné číslo

dochází až při přiřazení na výstup. Výsledná hodnota je pak přivedena na vstup žádané hodnoty regulátoru, stejně jako u první ukázky. Pouze už nedochází k odesílání dat o průběhu regulace, tudíž příslušné bloky už nejsou potřeba.



Obrázek 3.19: Schématické zapojení regulátoru DC motoru řízeného přes UART

3.8 Krokový motor

Na rozdíl od stejnosměrného motoru, není pro řízení krokového motoru potřeba generátor PWM. K jeho řízení je využita jednoduchá tabulka s definovanými hodnotami pro jednotlivé vodiče motoru. Každý řádek této tabulky obsahuje konfiguraci odpovídající jednomu stavu. Tabulka je navržena pro celé a poloviční kroky, nicméně by se dala upravit i pro jemnější kroky. Podle směru otáčení se jednotlivými stavy prochází nahoru nebo dolů. Průchodem přes všechny stavy se motor otáčí v polovičními kroky. Přepnutím na celé kroky se stavy pohybují po dvou.

STAV	A	B	C	D
0	1	0	0	1
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1

Tabulka 3.1: Logické hodnoty signálů pro různé stavy

Vstupy návrhu jsou opět hodinový signál, reset a datové vstupy pro rychlost a směr otáčení. Rychlost otáčení je v celých krocích za sekundu. Parametrem je velikost kroků.

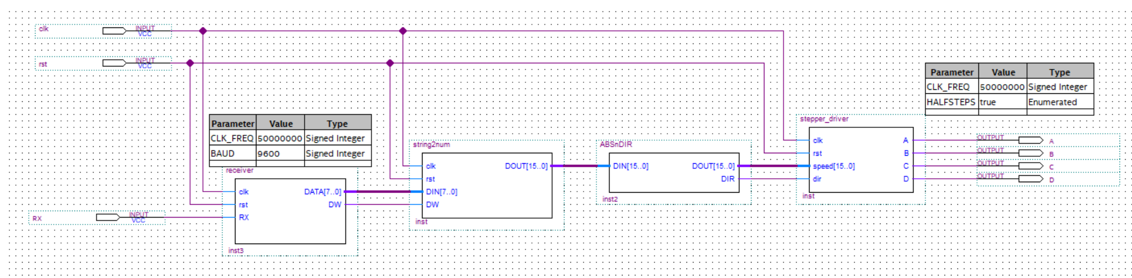
Tabulka je vytvořena obdobným způsobem, jako byla u vyrovnávací paměti. Opět se jedná o pole složené z vektorů. Vektory mají velikost 4 se sestupnými indexy a pole má velikost 8 se vzestupnými indexy. Následně jsou výčtem definovány všechny vektory, odpovídající tabulce 3.1.

```
type MEMORY is array (0 to 7) of std_logic_vector(3 downto 0);
constant TABLE : MEMORY := ("1001", "1000", "1100", "0100", "0110", "0010", "0011", "0001");
```

Pokud není návrh v resetu, kontroluje se zadaná rychlost. Pokud je nenulová, spustí se hlavní část návrhu. Podle nastavení velikosti kroků, celých nebo polovičních, se určí velikost zpoždění hodinového signálu. Jakmile dosáhne čítač hodnoty zpoždění, přiřadí se na signál *values* logické hodnoty signálů z tabulky, odpovídající současnému stavu. Následně se upraví hodnota stavu v závislosti na velikosti kroku a směru otáčení. Mimo proces jsou jednotlivé bity vektorového signálu *values* přiřazeny k příslušným výstupním signálům.

```
if HALFSTEPS = true then
        DELAY := CLK_FREQ / (2 * to_integer(speed));
else
        DELAY := CLK_FREQ / to_integer(speed);
end if;

count := count + 1;

if count >= DELAY then
        values <= TABLE(state);
        count := 0;

        if HALFSTEPS = true then
                if dir = '0' then
                        state := state + 1;
                else
                        state := state - 1;
                end if;
        else
                dir = '0' then
                        state := state + 2;
                else
                        state := state - 2;
                end if;
        end if;
end if;
```

Řízení motoru probíhá pomocí sériové komunikace UART z počítače. Součástí návrhu je tedy UART přijímač. Namísto vyrovnávací paměti je použit blok, který převádí ASCII symboly na číslo. Jedná se o stejný blok, jaký byl použit v předchozí úloze. Přijaté číslo se v dalším bloku rozloží na jeho absolutní hodnotu a znaménko. Tyto signály už vstupují přímo do bloku řídícího H-můstek s připojeným krokovým motorem.



Obrázek 3.20: Schématické zapojení řízení krokového motoru přes UART

3.9 Analogová váha

S analogovým převodníkem se komunikuje pomocní vlastního typu synchronní sériové komunikace. Řídící jednotka generuje hodinový signál a podle něj převodník přes jednu datovou linku posílá naměřenou hodnotu.

K převodu analogové hodnoty dochází při výchozím nastavení desetkrát za sekundu. Jakmile je převod hotový a hodnota je připravena k odeslání, nastaví převodník datový signál na log 0. V tu chvíli se spustí hodinový signál z hradlového pole. Následně je s každým taktem hodinového signálu odeslán jeden bit. Pro výchozí nastavení je analogová hodnota převedena na 24bitové číslo. Celý komunikační rámec je pak tvořen 24 cykly pro odeslání dat a jeden ukončovací, celkově tedy 25 cyklů.

Odesílaná data jsou ve formátu *two's complement.* Vzhledem k vysokému zesílení analogové hodnoty je pro další zpracování použito jen horních 16 bitů.

Obdobně jako UART přijímač je zpočátku komunikační blok ve stavu IDLE a vyčkává, než se na datovém signálu objeví log 0. Poté se IDLE změní na log 0 a začne se generovat hodinový signál *sck,* který zahájí komunikaci s převodníkem. Protože převodník odesílá bity s náběžnou hranou hodinového signálu, je vhodné data číst při seběžné hraně. V podmínce je napsáno *sck='1',* protože během spuštění procesu má signál *sck* skutečně hodnotu log 1 a až po ukončení procesu se změní na log 0. Pro prvních 24 průchodů dochází k zapisování příchozích bitů do přijímacího registru a následné zvyšování stavu. Následný průchod, ukončení přenosu, se stav vynuluje, IDLE se nastaví na log 1 a přijatá data se přiřadí na výstup. Mimo proces dochází k přiřazení interního hodinového signálu k výstupnímu signálu, připojenému k převodníku.

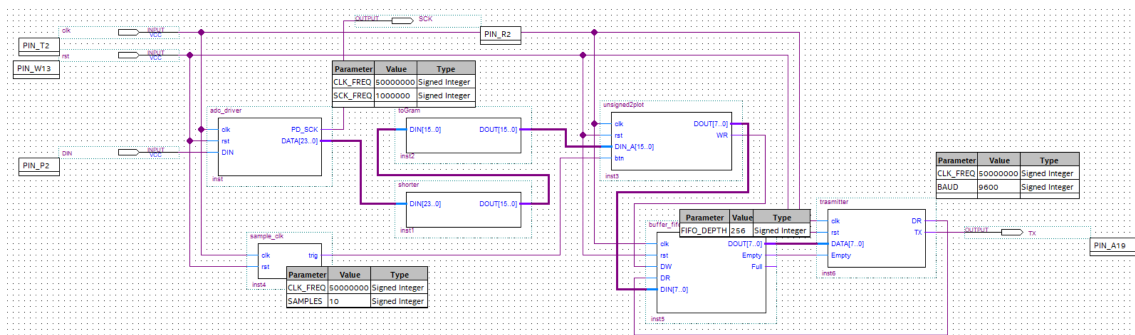
```
if idle = '0' then                                  -- probíhá čtení
        count := count + 1;

        if count >= DELAY then                      -- zpomalení na komunikační rychlost

                if sck = '1' then

                        if bit_state < 24 then      -- zápis 24 bitů
                                DATA_int(bit_state) := DIN;
                                bit_state := bit_state + 1;
```

```
                    else
                            bit_state := 0;       -- 25. hodinový impulz pro ukončení čtení dat
                            idle <= '1';
                            DATA <= DATA_int;
                    end if;
            end if;

            sck <= not sck;
            count := 0;
    end if;
end if;
```

Dalším prvkem je kalibrace váhy na nezatížení stav a přepočet na gramy. Struktura je obdobná jako u přepočtu tiků na otáčky za minutu v projektu s regulací stejnosměrného motoru. Konstanty výpočtu byly určeny experimentálně.

```
process(DIN)
    variable weight : integer := 0;
begin
    weight := to_integer(signed(DIN) + 560) * 10/13;
    DOUT <= std_logic_vector(to_signed(weight, 16));
end process;
```

Aby bylo možné číslo odeslat přes UART do počítače, nebo jiného zařízení, je nutné jej převést na ASCII znaky. K tomu je použit upravený blok z projektu regulace, který tentokrát převádí pouze jedno číslo. Následuje vyrovnávací paměť a UART odesílatel. Výstupem je změřená hmotnost uvedená v gramech vykreslovaná v reálném čase do grafu.



Obrázek 3.21: Schématické zapojení analogového převodníku s výstupem na sériový plotter

4 Závěr

Cílem této bakalářské práce bylo přinést bližší pohled do světa programovatelných hradlových polí a zájemcům o problematiku umožnit rychlejší a jednodušší cestu k jejich používání. Výsledkem by měla být schopnost čtenáře nejen pochopit a sestavit uvedené úlohy, ale s pomocí uvedených základů se rychleji dostat k návrhům plně využívajícím možnosti hradlového pole.

V úvodní části byla hradlová pole porovnána s mikroprocesory, které dnes můžeme najít úplně všude. Byly zmíněny některé jejich výhody, nevýhody a příklady využití. Následně byla uvedena historie vývoje a cesta vedoucí k modelu použitém pro zpracování uvedených úloh. Po seznámení s kroky vedoucími k současné architektuře, v rámci použitého hardwaru, byla představena vnitřní struktura hradlových polí a popsány funkce dílčích prvků. Bylo také zmíněno, jak se hradlová pole programují. Další teoretická část byla věnována základní teorii potřebné k vypracování samotných úloh.

Začátkem praktické části byl představen použitý hardware, ke kterému se vztahovala některá teorie. Stejně tak došlo k představení a popisu vývojového prostředí, ve kterém byly úlohy vypracovány. Hlavní části této práce je vypracování samotných úloh. Z počátku ne zcela související příklady, jako sériová komunikace UART a generátor PWM, se postupně spojují do tří větších projektů, kterými jsou regulace stejnosměrného motoru, řízení krokového motoru a váha s analogovým převodníkem. Dílčí části návrhů jsou psány v jazyce VHDL a posléze spojeny v blokovém schématu.

Vytvořené projekty rozhodně nemají představovat nejlepší možné řešení. Naopak se v některých případech jedná o co nejjednodušší řešení, a to proto aby bylo snadné je nejen sestavit, ale hlavně i pochopit bez příliš rozsáhlých znalostí. Cílem této práce není dokonale pochopit protokoly sériových komunikací nebo konstrukce krokových motorů, ale seznámit s hradlovými poli, práce s popisovacími jazyky a samotný postup práce od vytvoření nového projektu až po jeho nahrání do zařízení.

Literatura

[1] TRIMBERGER, Stephen M. *Proceedings of the IEEE* [online]. Vol. 103, No. 3, March 2015 [cit. 2020-05-02]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7086413

[2] Basics of FPGA Architecture and Applications. *elprocus.com* [online]. [cit. 2020-05-02]. Dostupné z: https://www.elprocus.com/fpga-architecture-and-applications/

[3] BISWAS, Priyabrata. Introduction to FPGA and its Architecture. *towardsdatascience.com* [online]. November 20, 2019 [cit. 2020-05-02]. Dostupné z. https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c

[4] VHDL. *Wikipedia* [online]. [cit. 2020-05-02]. Dostupné z: https://en.wikipedia.org/wiki/VHDL

[5] PELLERIN, David. An Introduction to VHDL. *uco.es* [online]. [cit. 2020-05-02]. Dostupné z: http://www.uco.es/~ff1mumuj/h_intro.htm

[6] KEIM, Robert. Back to Basics: The Universal Asynchronous Receiver/Transmitter (UART). *allaboutcircuits.com* [online]. December 20, 2016 [cit. 2020-05-02]. Dostupné z: https://www.allaboutcircuits.com/technical-articles/back-to-basics-the-universal-asynchronous-receiver-transmitter-uart/

[7] Universal asynchronous receiver-transmitter. *Wikipedia* [online]. [cit. 2020-05-02]. Dostupné z: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[8] Pulzně šířková modulace. *Wikipedia* [online]. [cit. 2020-05-02]. Dostupné z: https://cs.wikipedia.org/wiki/Pulzn%C4%9B_%C5%A1%C3%AD%C5%99kov%C3%A1_modulace

[9] Rotační enkodéry ELTRA. *odbornecasopisy.cz* [online]. [cit. 2020-05-02]. Dostupné z: http://www.odbornecasopisy.cz/elektro/casopis/tema/rotacni-enkodery-eltra--13671

[10] Magnetic Encoders. *dynapar.com* [online], [cit. 2020-05-02]. Dostupné z: https://www.dynapar.com/Technology/Encoder_Basics/Magnetic_Encoder/

[11] Stejnosměrné motory. *Wikipedia* [online]. [cit. 2020-05-02]. Dostupné z: https://cs.wikipedia.org/wiki/Stejnosm%C4%9Brn%C3%BD_motor

[12] KELLINGHUSEN, Martin. Choosing and integrating a medical device stepper motor. *starfishmedical.com* [online]. [cit. 2020-05-02]. Dostupné z: https://starfishmedical.com/blog/medical-device-stepper-motor/

[13] Stepper motor. *Wikipedia* [online]. [cit. 2020-05-02]. Dostupné z: https://en.wikipedia.org/wiki/Stepper_motor

[14] VODA, Zbyšek. Rychlé seznámení s PID regulátorem. *arduino.cz* [online]. [cit. 2020-05-02]. Dostupné z: https://arduino.cz/rychle-seznameni-s-pid-regulatorem/

[15] FTDI232 – Google Sites. *sites.google.com* [online]. [cit. 2020-05-02]. Dostupné z: https://sites.google.com/site/cisc071jc/_/rsrc/1508961406913/public/topics/ftdi232/ftdi232pinout.gif?height=182&width=400

[16] CieNTi/serial_port_plotter. *GitHub* [online]. [cit. 2020-05-02]. Dostupné z: https://github.com/CieNTi/serial_port_plotter

[17] Quartus Prime. *Intel* [online]. [cit. 2020-05-02]. Dostupné z: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[18] Vážící senzor 5kg + ADC modul HX711. *laskarduino* [online]. [cit. 2020-05-22]. Dostupné z: https://lasklab.cz/img/cms/HX711-connection.jpg

[19] UART Explained. *electric imp* [online]. [cit. 2020-05-22]. Dostupné z: https://developer.electricimp.com/sites/default/files/attachments/images/uart/uart3.png

[20] KRÁL, Jiří. Řešené příklady ve VHDL: Hradlová pole FPGA pro začátečníky. Praha: BEN - technická literatura, 2010, 127 s. ISBN 978-807-3002-572

[21] PANDIT, Abhiemanyu. Introduction to FPGA and It's Programming Tools. *CircuitDigest* [online]. [cit. 2020-05-22]. Dostupné z: https://circuitdigest.com/sites/default/files/inlineimages/u1/FPGA-Structure.png

[22] Field-Programmable gate arrays. *Wikipedia* [online]. [cit. 2020-05-22]. Dostupné z: https://en.wikipedia.org/wiki/Field-programmable_gate_array

[23] PID controller. *Wikipedia* [online]. [cit. 2020-05-22]. Dostupné z: https://en.wikipedia.org/wiki/PID_controller#/media/File:PID_en.svg

[24] Pulse Width Modulation. *Electronics Tutorials* [online]. [cit. 2020-05-22]. Dostupné z: https://www.electronics-tutorials.ws/wp-content/uploads/2018/05/articles-pwm3.gif

[25] Basics of FPGA Architecture and Applications. *elprocus.com* [online]. [cit. 2020-05-02]. Dostupné z: https://www.elprocus.com/wp-content/uploads/2014/09/9-15-2014-7-03-24-PM.jpg

[26] KELLINGHUSEN, Martin. Choosing and integrating a medical device stepper motor. *starfishmedical.com* [online]. [cit. 2020-05-02]. Dostupné z: https://starfishmedical.com/assets/Unipolar-vs-Bipolar-Configuration-300x191.png

[27] Rotační enkodéry ELTRA. *odbornecasopisy.cz* [online]. [cit. 2020-05-02]. Dostupné z: http://www.odbornecasopisy.cz/imagesold/e0405522.gif

[28] DAHL, Øyvind Nydal. What is an H-Bridge?. *build electronic circuits* [online]. December 5, 2018 [cit. 2020-05-22]. Dostupné z: https://www.build-electronic-circuits.com/wp-content/uploads/2018/11/H-bridge-full-1024x562.png

[29] Rotary Encoder. *nxr.northwestern.edu* [online]. [cit. 2020-05-22]. Dostupné z: http://hades.mech.northwestern.edu/images/2/22/Encoder_diagram.png

[30] COLLINS, Danielle. What stepper motor type is best for high torque?. *motioncontroltips.com* [online]. [cit. 2020-05-22]. Dostupné z: https://3l4sbp4ao2771ln0f54chhvm-wpengine.netdna-ssl.com/wp-content/uploads/2017/05/Oriental-Hybrid-Stepper-Motor-Construction.jpg

[31] Modeling the Pulse-Width Modulator. allaboutcicuits.com [online]. June 03, 2015 [cit. 2020-05-22]. Dostupné z: https://www.allaboutcircuits.com/uploads/articles/9A-Modelling-the-Pulse-Width-Modulator_(2).png

[32] ChinaQMTECH/CYCLONE_IV_EP4CE15. *GitHub* [online]. [cit. 2020-05-22]. Dostupné z: https://github.com/ChinaQMTECH/CYCLONE_IV_EP4CE15

[33] Rotary Encoder with Push Switch. *addicore.com* [online]. [cit. 2020-05-22]. Dostupné z: https://cdn3.volusion.com/btfzd.umflq/v/vspfiles/photos/AD267-2T.jpg

[34] Quadrature encoder signal from dc motor is very noisy. *Stack Exchange* [online]. [cit. 2020-05-22]. Dostupné z: https://i.stack.imgur.com/sxUXp.png

Seznam použitých zkratek

FPGA	Field-Programmable Gate Array
ASIC	Application Specific Integrated Circuit
EDA	Electronic Design Automation
EPROM	Erasable Programmable Read-Only Memory
SRAM	Static Random Access Memory
HDL	Hardware Description Language
IOB	Input/Output block
LUT	Look-Up Table
MUX	Multiplexer
VHSIC	Very High Speed Integrated Circuit
UART	Universal Asynchronous Receiver-Transmitter
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
PWM	Pulse Width Modulation
DC	Direct Current
PID	Proportional–Integral–Derivative
PSD	Proportional-Sum-Derivative
USB	Universal Serial Bus
FIFO	First In First Out
ASCII	American Standard Code for Information Interchange
LF	Line Feed

Seznam příloh

Přílohy na SD kartě

Každý složka obsahuje všechny soubory potřebné k sestavení projektu. Soubory s příponou *bdf* jsou vrchní schematické návrhy. Dílčí části projektu jsou ve dvou typech souborů. Prvním typem je *vhd* soubor, který obsahuje samotný kód v jazyce VHDL. Soubory typu *bsf* jsou symboly, vytvořené ze souborů *vhd*, které jsou použity ve schematickém návrhu.

```
| —— 01_UART
        | —— buffer_fifo.bsf
        | —— buffer_fifo.vhd
        | —— debouncer_short.bsf
        | —— debouncer_short.vhd
        | —— demo_echo.bdf
        | —— demo_fifo.bdf
        | —— receiver.bsf
        | —— receiver.vhd
        | —— transmitter.bsf
        | —— transmitter.vhd
| —— 02_PWM
        | —— PWM_gen.bsf
        | —— PWM_gen.vhd
| —— 03_Encoder
        | —— demo_enk,bdf
        | —— encoder_signed.bsf
        | —— encoder_signed.vhd
        | —— encoder_unsigned.bsf
        | —— encoder_unsigned.vhd
        | —— PWM_gen.bsf
        | —— PWM_gen.vhd
| —— 04_PID
        | —— buffer_fifo.bsf
        | —— buffer_fifo.vhd
        | —— encoder_signed.bsf
        | —— encoder_signed.vhd
        | —— encoder_speed.bsf
        | —— encoder_speed.vhd
        | —— motor_uart.bdf
        | —— PID.bsf
        | —— PID.vhd
        | —— pid_plotter.bdf
        | —— plotter2_signed16.bsf
        | —— plotter2_signed16.vhd
        | —— PWM_gen.bsf
        | —— PWM_gen.vhd
        | —— receiver.bsf
        | —— receiver.vhd
```

```
          |—— sample_clk.bsf
          |—— sample_clk.vhd
          |—— string2num.bsf
          |—— string2num.vhd
          |—— ticks2rpm.bsf
          |—— ticks2rpm.vhd
          |—— transmitter.bsf
          |—— transmitter.vhd
|—— 05_Stepper
          |—— ABSnDIR.bsf
          |—— ABSnDIR.vhd
          |—— demo_stepper.bdg
          |—— receiver.bsf
          |—— receiver.vhd
          |—— stepper_driver.bsf
          |—— stepper_driver.vhd
          |—— string2num.bsf
          |—— string2num.vhd
|—— 06_ADC
          |—— adc_driver.bsf
          |—— adc_driver.vhd
          |—— buffer_fifo.bsf
          |—— buffer_fifo.vhd
          |—— demo_vaha.bdf
          |—— sample_clk.bsf
          |—— sample_clk.vhd
          |—— shorter.bsf
          |—— shorter.vhd
          |—— toGram.bsf
          |—— toGram.vhd
          |—— transmitter.bsf
          |—— transmitter.vhd
          |—— unsigned2plot.bsf
          |—— unsigned2plot.vhd
```