Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Vývoj zařízení Modbus TCP na technologii FPGA

Diplomová práce

květen 2016 Bc. Miroslav Dvořák, DiT.

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2014/2015**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Miroslav Dvořák, Dipl.tech.**
Osobní číslo:	**I13403**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vývoj zařízení Modbus TCP na technologii FPGA**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je navrhnout a sestrojit zařízení Modbus TCP na technologii FPGA. Na hradlovém poli bude realizován soft core procesor, který bude komunikovat přes rozhraní Ethernet. Zařízení bude napájeno stejnosměrným napětím 24 V a bude obsahovat minimálně 8 digitálních vstupů, 8 digitální výstupů, 4 analogové vstupy a LCD displej zobrazující stavové hodnoty. Realizované zařízení bude otestováno v síti Ethernet s programovatelným automatem WAGO-I/O-SYSTEM 750 a bude také vytvořena ukázková úloha pro inteligentní řízení budovy.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

ŠMEJKAL, Ladislav. PLC a automatizace 1: Základní pojmy, úvod do programování. Praha: BEN - technická literatura, 2002. ISBN 80-86056-58-9.
ŠMEJKAL, Ladislav. PLC a automatizace 2: Sekvenční logické systémy a základy fuzzy logiky. Praha: BEN - technická literatura, 2005. ISBN 80-7300-087-3.
GARLÍK, Bohumír. Inteligentní budovy. Praha: BEN - technická literatura, 2012. ISBN 978-80-7300-440-8.
KOLOUCH, Jaromír. Jazyk Verilog a jeho užití při modelování a syntéze číslicových systémů. Brno: Vydavatelství Akademické nakladatelství, VUTIUM, 2012. ISBN 9788021445161.

Vedoucí diplomové práce: **Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2014**
Termín odevzdání diplomové práce: **15. května 2015**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2014

Prohlášení autora

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 1. 5. 2016 Bc. Miroslav Dvořák, DiT.

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Michaelu Bažantovi, PhD. za cenné rady a připomínky, které mi poskytl během vypracování mé diplomové práce, a za jeho ochotný přístup.

Anotace

Tato diplomová práce se zabývá návrhem a realizací zařízení Modbus TCP na technologii FPGA. Vlastní řešení spočívá v realizaci soft-procesoru a všech požadovaných periferií na hradlovém poli. Následuje vytvoření řídící aplikace pro tento soft-procesor. Výsledný modul je propojen s programovatelným automatem Wago řady 750 a na cvičných úlohách je otestována jeho činnost.

Klíčová slova

ALTERA, CODESYS, ETHERNET, FPGA, MODBUS, NIOS , PLC, WAGO

Annotation

This thesis describes the design and implementation of Modbus TCP devices on FPGA technology. Contribution consists in the realization of soft-processor and all the required peripherals on the FPGA.On top of this was created control application for the soft-processor. The resulting module is connected to a programmable logic controller Wago 750 series, and it was tested with training tasks.

Keywords

ALTERA, CODESYS, ETHERNET, FPGA, MODBUS, NIOS , PLC, WAGO

OBSAH

SEZNAM ZKRATEK

ADU	Application Data Unit
ADC	Analog-to-Digital Converter
ARM	Advanced RISC Machine
BCD	Binary Coded Decimal
CPLD	Complex Programmable Logic Device
FBD	Function Block Diagram
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GTK	Gimp ToolKit
GUI	Graphical User Interface
HDL	Hardware Description Language
IDE	Integrated Development Environment
IOB	Input Output Block
IP	Internet Protocol
IRQ	Interrupt ReQuest
JTAG	Joint Test Action Group
LB	Logic Block
LCD	Liquid Crystal Display
LED	Light Emitting Diode
NC	Not Connected
OOP	Object-Oriented Programming
PCB	Printed Circuit Board
PDU	Protocol Data Unit
PLC	Programmable Logic Controler
PLD	Programmable Logic Device
SoC	System on a Chip
SPI	Serial Peripheral Interface
ST	Structured text
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver
USB	Universal Serial Bus
UDP	User Datagram Protocol
VCD	Value Change Dump
XOR	eXlusive OR

SEZNAM OBRÁZKŮ

1 Úvod

Cílem diplomové práce je navrhnout a sestrojit zařízení Modbus TCP na technologii FPGA. Toto řešení oproti klasickému přidávání rozšiřujících karet ke stávajícímu systému PLC je sice na první pohled méně variabilní, ale v konečném řešení podstatně snižuje náklady a mnohdy zjednodušuje vlastní řízení technologie. Snížení nákladů je způsobeno implementací požadovaných rozhraní na čipu FPGA, nutná je pouze realizace galvanického oddělení signálů. V případě, kdy zákazník požaduje velké množství rozdílných komunikačních rozhraní, je cenový rozdíl podstatný, zanedbáme-li čas vlastního vývoje.

Pro komunikaci je záměrně použit jednoduchý komunikační protokol Modbus, se kterým umí pracovat většina programovatelných automatů. Varianta protokolu Modbus TCP využívá běžnou ethernetovou síť a je tedy vhodná i pro decentralizaci řízení procesů.

Vlastní řešení spočívá v realizaci soft-procesoru a všech požadovaných periferií na hradlovém poli. Následuje vytvoření řídící aplikace pro tento soft-procesor. Zde můžeme využít zmíněného principu decentralizace a část technologie řídit pomocí tohoto modulu.

Pro kompatibilitu s PLC je modul navržen pro 24V logiku s galvanickým oddělením všech vstupů a výstupů. Integrovaný LCD displej zobrazuje stavové a chybové informace.

Realizované zařízení bude v závěru otestováno společně s programovatelným automatem WAGO-I/O-SYSTEM 750. Pro otestování budou vytvořeny tři ukázkové úlohy řešící vzájemnou komunikaci digitálních vstupů, digitálních výstupů a analogových vstupů s PLC.

2 PLC

Programovatelný logický automat neboli PLC (Programmable Logic Controller) je relativně malý počítač používaný pro automatizaci procesů v reálném čase – řízení strojů nebo výrobních linek. Pro PLC je charakteristický nejen zvětšený rozsah provozních teplot, ale i to, že program vykonává v cyklech (čtení vstupů - zpracování dat - zápis výstupů - režie PLC) a jeho periferie jsou přímo uzpůsobeny pro napojení na technologické procesy. K programování PLC systémů se používají specializované jazyky (norma IEC 1131-3) a to buď v textové (IL - jazyk mnemokódů, ST - jazyk strukturovaného textu) nebo grafické (LD - jazyk reléových schémat, FBD - jazyk funkčních bloků).

Z hlediska konstrukce PLC se dělí na dvě základní skupiny:

- *kompaktní systém* - V jednom modulu obsahuje CPU (Central Procesor Unit), analogové a digitální vstupy/výstupy, komunikaci a většinou i zdrojovou část. Rozšiřitelnost těchto systémů je velmi omezena.
- *modulární systém* - Jednotlivé komponenty PLC jsou rozděleny do samostatných výměnných modulů (zdroj, CPU, vstupy/výstupy, časovače, apod.). Tento systém je možno dále rozšiřovat nejen do počtu v/v modulů, ale i z pohledu zvyšování výkonu.

Obrázek 1 – Modulární systém Wago. Zdroj [1].

Jednotlivé funkční moduly PLC jsou rozděleny do kategorií, viz [14] a v případě, že projektant potřebuje ovládat systémem 10 digitálních výstupů, dále potřebuje číst 4 analogové výstupy a ještě k tomu potřebuje sériovou komunikaci a časovač, kompaktní systém mu vždy nemusí plně vyhovovat. Proto zvolí modulární systém, zde však jednotlivé moduly nejsou plně využity a neúměrně narůstá cena celého řídícího systému.

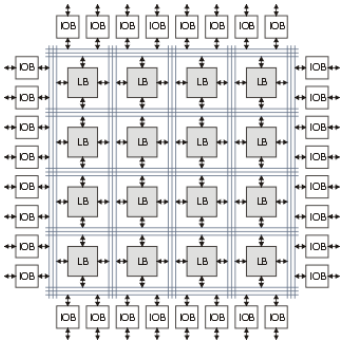
Diplomová práce si proto klade za cíl navrhnout a sestrojit zařízení Modbus TCP na hradlovém poli, které umožní jednoduše a levně rozšířit PLC (Programmable Logic Controller) o další v/v jednotky. Jednotlivé požadované moduly se budou vytvářet za použití jazyka HDL, což umožňuje na stejném hardwaru vytvořit odlišný systém. Výhodou zvoleného protokolu Modbus TCP je to, že ho zná většina programovatelných automatů, což zvyšuje univerzálnost navrhovaného řešení.

Pro účely testování bude systém obsahovat pouze 16 digitálních vstupů, 16 digitálních výstupů, 4 analogové vstupy a stavový LCD displej. Digitální výstupy budou rozděleny na 12 reléových a na 4 tranzistorové s otevřeným kolektorem, pro generování rychlých průběhů, např. PWM.

3 PROGRAMOVATELNÁ HRADLOVÉ POLE

Programovatelná hradlová pole (FPGA, Field Programmable Gate Array) jsou speciální číslicové integrované obvody obsahující různě složité programovatelné bloky propojené konfigurovatelnou maticí spojů umožňující implementaci od enkodérů/dekodérů až po složité soft-procesory (např. ARM).

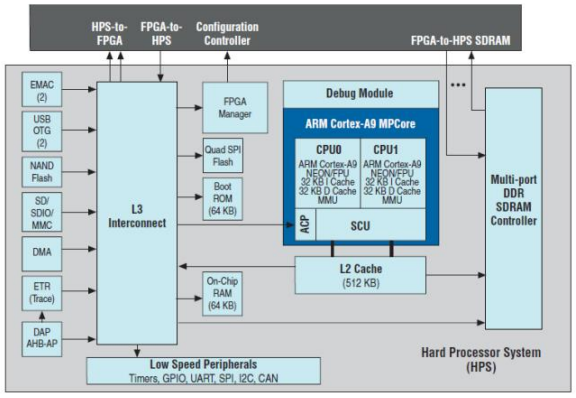
Typickou strukturu FPGA obvodu znázorňuje obrázek č. 2. Bloky označené IOB (Input/Output Block) představují vstupně-výstupní obvody pro každý v/v pin FPGA. Tyto bloky obvykle obsahují registr, budič, multiplexer a ochranné obvody. Bloky LB (Logic Block) představují vlastní programovatelné logické bloky.



Obrázek 2 - Struktura obvodu FPGA. Zdroj [9].

Soft-procesor je realizován pomocí jazyka HDL v obvodech obsahujících programovatelnou logiku a na rozdíl od klasických procesorů realizovaných technologickým postupem na křemíku lze jeho strukturu libovolně modifikovat.

FPGA obvody dnes nacházejí uplatnění v široké škále aplikací díky své programovatelnosti, snadnému návrhu a flexibilitě. V současné době jsou stále více populární obvody typu SoC, které integrují v jednom pouzdru procesor ARM a FPGA.



Obrázek 3 - SoC Cyclone V od firmy Altera. Zdroj [6].

3.1 ALTERA

Firma Altera Corporation vznikla v roce 1983 a je v současnosti druhý největším producentem hradlových polí na světě. První místo patří firmě Xilinx. Hlavní sídlo se nachází v San Jose, USA. Od okamžiku založení byla průkopníkem v návrhu a výrobě programovatelných logických obvodů. Ve své nabídce má v současnosti obvody FPGA, SoC, CPLD, včetně kompletní sady vývojových nástrojů.

Obrázek 4 - Logo Altera Corporation. Zroj [6].

V létě 2015 koupil největší světový výrobce čipů INTEL konkurenční Alteru za 16,7 miliard dolarů. Spojením tak bude Intel moci prodávat nejen programovatelné čipy z produkce Altery, ale i chystá integraci technologie Altery do procesorů Xeon a do nových čipů, např. pro internet věcí.

3.2 TERASIC

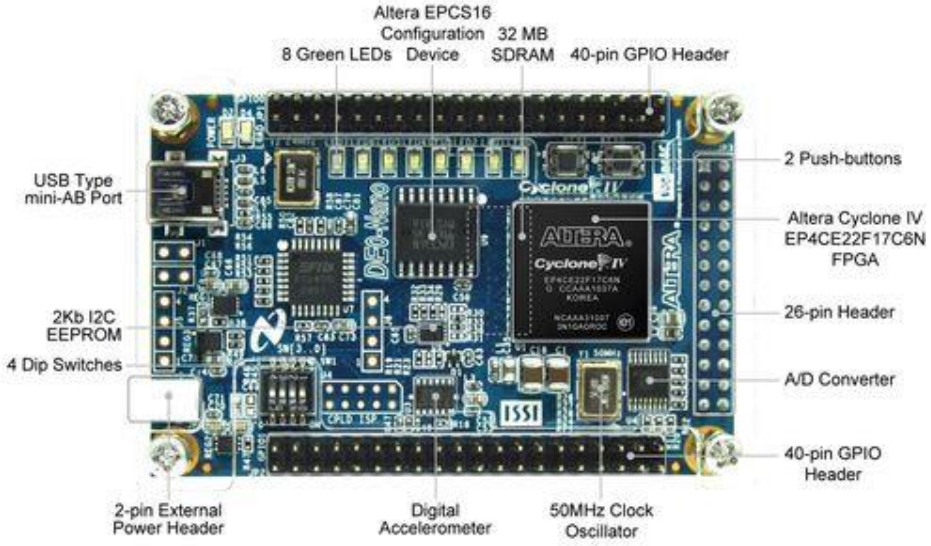
Firma Terasic byla založena v roce 2000 s hlavním cílem rozvíjet vývojové platformy pro FPGA. Hlavní centrum se nachází v Hsin Chu na Tchaj-wanu, které je známé jako asijské Silicon Valley. Firma je především zaměřena na čipy Altera, u kterých nabízí široké spektrum vývojových kitů včetně kvalitní uživatelské podpory.

Obrázek 5 - Logo Terasic Inc. Zdroj[10].

3.3 MODUL DE0-NANO

Modul od firmy Terasic Inc. patří do kategorie levných vývojových kitů a je osazen obvodem FPGA Cyclone IV od firmy Altera. Modul je zejména vhodný pro účely vývoje, výroby funkčních prototypů apod. Primárně je zaměřen pro embedded aplikace, proto modul obsahuje integrovaný programátor USB-Blaster, paměti SDRAM, EEPROM, ADC, 3-osý akcelerometr, LED diody, tlačítka a především dvě 40pinové lišty s GPIO. S kitem jsou také dodávány dvě aplikace, první pro otestování všech komponent na kitu, druhá pro generování šablony pro top-level návrh (přiřazení pinu apod.). Modul obsahuje vše

potřebné nejen pro implementaci 32-bitového soft-procesoru NIOS, ale i pro realizaci zadání této diplomové práce.



Obrázek 6 - Modul DE0-Nano. Zdroj [10].

4 VERILOG

Jazyk Verilog patří do skupiny jazyků HDL (Hardware Description Language), které se používají pro návrh aplikací obvodů CPLD a FPGA. Základní verze tohoto jazyka byla přijata jako standard IEEE v roce 1995. Je to druhý nejrozšířenější jazyk po VHDL a je primárně určen pro design, verifikaci a realizaci digitálních obvodů.

Vzhledem k podobné syntaxi jazyka Verilog k jazyku C nebude jazyk popsán podrobně, více informací poskytuje zdroj [2], který je v současné době jako jediný dostupný v českém jazyce.

Základním prvkem jazyka Verilog je programový blok tzv. modul. V modulu najdeme deklaraci jeho vstupů a výstupů pro komunikaci s okolím. Dále jsou v něm definovány vnitřní signály, promněnné, konstanty a funkce. Na obrázku č.7 je zobrazen modul jednoduchého binárního čítače. Vstupní signály jsou označeny input, výstupní signály output. V jazyce Verilog rozeznáváme dva typy dat: propojení a promněnné. Propojení označujeme *wire* a můžeme si ho představit jako vodič. Naproti tomu promněnné si mohou uchovávat svoji hodnotu až do následující změny dalším procedurálním příkazem.

V každém modulu najdeme minimálně jeden blok označený direktivou always. Jeho parametr nám určuje, kdy bude provedeno tělo tohoto bloku. V našem případě se bude provádět při každé vzestupné hraně (posedge) hodinového signálu. Tělo bloku obsahuje podmínku, kde testujeme hodnotu vstupního signálu reset. V případě, že signál není aktivní, je provedena inkrementace výstupní hodnoty. V opačném případě je hodnota výstupu nastavena na nulu, podmínkou je realizován tzv. synchronní reset.

```
module binary_counter    (
input wire clk,            // vstup hodinového signálu
input wire reset,          // vstup reset čítače
output reg [7:0] out   //výstup čítače
);

always @(posedge clk)
begin
 if (reset)
    out <= 8'b0 ;
  else
    out <= out + 1;
end
endmodule
```

Obrázek 7 - Kód binárního čítače.

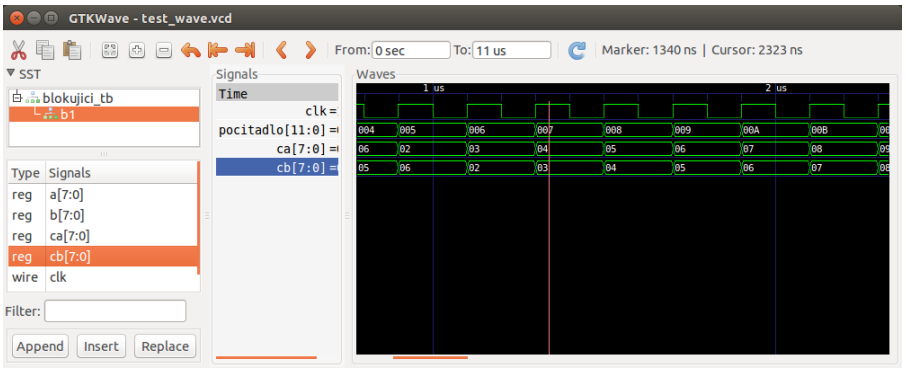
4.1 ICARUS VERILOG

Icarus Verilog je open-source nástroj pro simulaci a syntézu. Hlavní překladač jazyka napsal Stephen Williams. Icarus Verilog je portován především na linuxové systémy, jako nástroj pro příkazovou řádku. Nástroj nainstalujeme pomocí příkazu *sudo apt-get install iverilog*.

Velkou výhodou Icarus Verilog je především rychlost návrhu a verifikace jednotlivých verilog modulů bez nutnosti instalace objemného vývojového prostředí.

4.2 GTKWAVE

GTKWave je plně funkční GTK + prohlížeč průběhů, který čte FST, LXT, LXT2, VZT, GHW soubory, včetně standardních Verilog VCD / EVCD, viz obr. 8. GTKWave je primárně vyvíjen pro OS Linux, existují však také porty pro další operační systémy včetně Microsoft Windows (buď jako nativní aplikace Win32 nebo přes Cygwin). GTKWave je součástí open-source projektu gEDA.



Obrázek 8 - Program GTKWave.

4.3 TVORBA MODULU PRO TEST BENCH

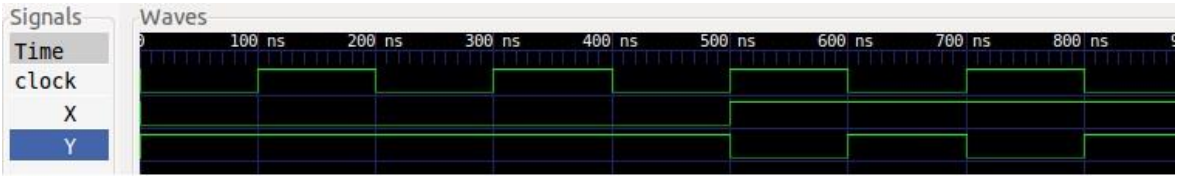
Jak už název napovídá, budeme programovat jednoduchý testovací bench pro kontrolu našeho vytvořeného modulu. Základem je standardní modul, který nemá žádné vstupní ani výstupní signály, viz obrázek č. 9.

```
`timescale 100 ns/1 ps

module test_tb();
reg clock=0;
reg X = 0;
wire Y;

initial begin
  $dumpfile ("test_wave.vcd");
  $dumpvars;
  #5 X = 1;
  #5 $finish;
end

always begin
  #1 clock = !clock;
end

nand  n1(Y,clock,X);
endmodule
```

Obrázek 9 - Ukázka Verilog kódu.

Obsahuje pouze vnitřní registry a vodiče potřebné pro testování zvoleného modulu. Základem je registr clock simulující hodinový signál. Jeho logická úroveň je vždy změněna v každé periodě vnitřního simulačního času a je dána direktivou „#1“. Pokud chceme měnit vstupní simulované signály, opět použijeme zmíněnou direktivu a zapíšeme za ní název proměnné a její novou hodnotu. Například pokud chceme u proměnné změnit hodnotu za 5 cyklů simulačního času, zapíšeme následující příkaz „#5 X=1“.

Na konci modulu je vytvořena instance n1 testovaného modulu nand(výstup, vstup, vstup) a zadány potřebné vstupní, resp. výstupní proměnné.



Obrázek 10 - Ukázka simulace.

4.4 VERILOG V PŘÍKLADECH

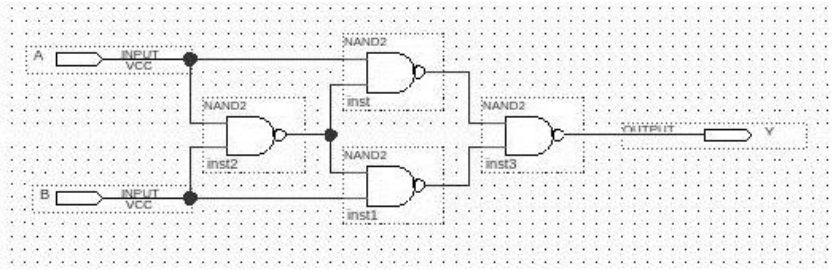
V následující kapitole budou probrány základy jazyka Verilog na příkladech. U každého příkladu je provedena simulace s popisem. Výstupem simulace může být i soubor vcd (Value Change Dump) obsahující simulované průběhy signálů. Tyto soubory lze zobrazit například pomocí zmíněného programu GTKWave. První ukázka bude realizovat hradlo XOR pomocí hradel NAND a druhá bude realizovat posuvný registr. V obou příkladech bude obvod vytvořen pomocí schématických symbolů a pomocí přímého zápisu v jazyce Verilog. Další příklady již ukazují praktické využití.

4.4.1 REALIZACE HRADLA XOR POMOCÍ HRADEL NAND

Hradlo XOR (eXclusive OR) je jedním ze základních kombinačních logických obvodů, jehož výstup je exkluzívní logický součet vstupů. Výstup je log. 1 pouze pokud se hodnoty vstupů liší, pravdivostní tabulka a výsledné náhradní schéma je na obrázku č. 11.

Logická funkce je vyjádřena vztahem $Y=A \oplus B=A \bullet B'+A' \bullet B$ (logický součin je označen znakem •, negace je označena znakem ') a lze ji s využitím Booleovy algebry a De Morganových zákonů přepsat do tvaru $Y=((A \bullet (A \bullet B)') \bullet (B \bullet (A \bullet B)'))'$.

VSTUPY		VÝSTUP
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0



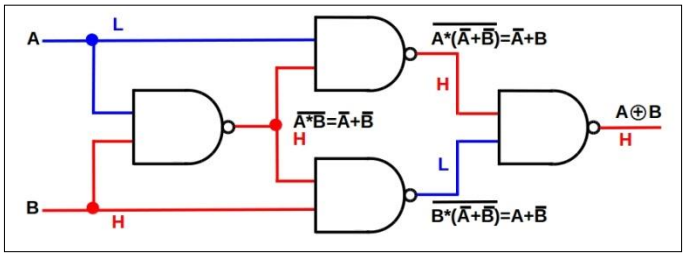
Obrázek 11 - Pravdivostní tabulka a náhradní schéma hradla XOR.

Náhradní schéma logického hradla XOR je nakresleno v editoru schémat v programu Quartus a následně převedeno na Verilog soubor (File->Create HDL design from current file, soubor xor_sch.v). Druhý způsob je zápis funkce s využitím předdefinovaných hradel do Verilog souboru *xor_verilog1.v*. Třetí způsob je zápis funkce za použití Verilog operátorů, viz soubor *xor_verilog2.v*.

```
module test;

  /* vytvori signaly na vstupu  */
  reg A = 0;
  reg B = 0;
  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,test);
    $timeformat(-3, 0, " ms", 6);

    # 10 A = 1;
    # 10 B = 1;
    # 10 A = 0;
    # 10 B = 0;
    # 10 $finish;
  end

  wire Y1,Y2,Y3,Y4;

  //XOR s vyuzitim operatoru (vzor)
  assign Y1=A^B;

 //XOR s vyuzitim schematickych znacek
  XOR_sch xorsch(A, B,Y2);
```

Obrázek 12 - Ukázka části kódu testovacího souboru xor_tb.v.

Následuje vytvoření testovacího souboru xor_tb.v, který je složen ze 4 částí. První část využívá přímo operátor pro logickou funkci XOR, druhá část testuje modul vytvořený ze schématu, třetí část testuje soubor přímo psaný v jazyce Verilog za použití již předdefinovaných hradel a čtvrtá část využívá logické operátory jazyka Verilog. Všechny vytvořené průběhy jsou zobrazeny pro vzájemné srovnání v programu GTKWave.



Obrázek 13 - Ukázka schématu hradla XOR. Zdroj [5].

Pořadí jednotlivých příkazů volaných v příkazové řádce:

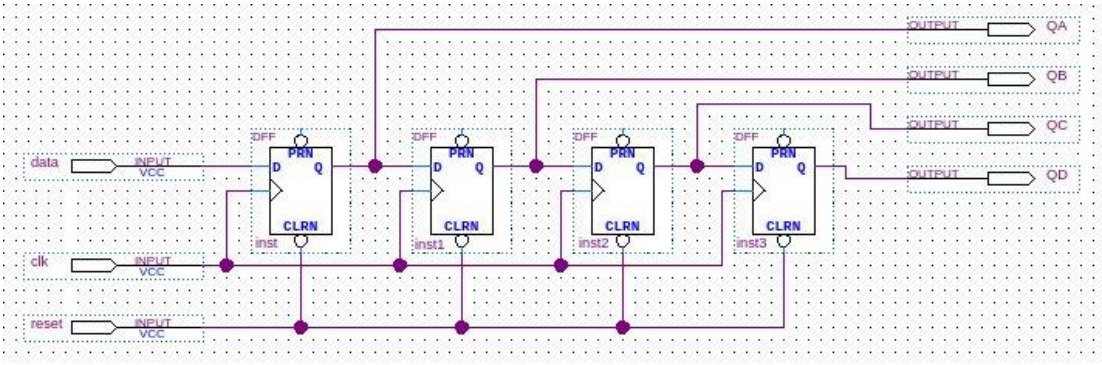
- zkompilujeme jednotlivé moduly příkazem *iverilog -o xor XOR_tb.v XOR_sch.v XOR_verilog1.v XOR_verilog2.v*,
- provedeme simulaci (textový výpis) příkazem vvp xor,
- jednotlivé průběhy ze simulace graficky porovnáme v programu GTKWave příkazem *gtkwave test.vcd &*.

Obrázek 14 - Zobrazení průběhu simulace v programu GTKWave.

Výsledky realizované simulace potvrdily možnost využití tří typů zápisu pro realizovanou funkci. Nejjednodušší a zároveň nejrychlejší je zápis poslední, tj. funkce je zapsána pomocí logických operátorů jazyka Verilog. Podrobnější popis konstrukce testovacího souboru je popsán v odkazu [16].

4.4.2 REALIZACE 4-BITOVÉHO POSUVNÉHO REGISTRU

Posuvný registr je skupina klopných obvodů, která má propojené vstupy a výstupy tak, že s náběžnou hranou hodinového signálu jsou data (bity) synchronně posunuty o jeden klopný obvod. Na tomto principu pracuje například sériová linka RS 232 nebo SPI.

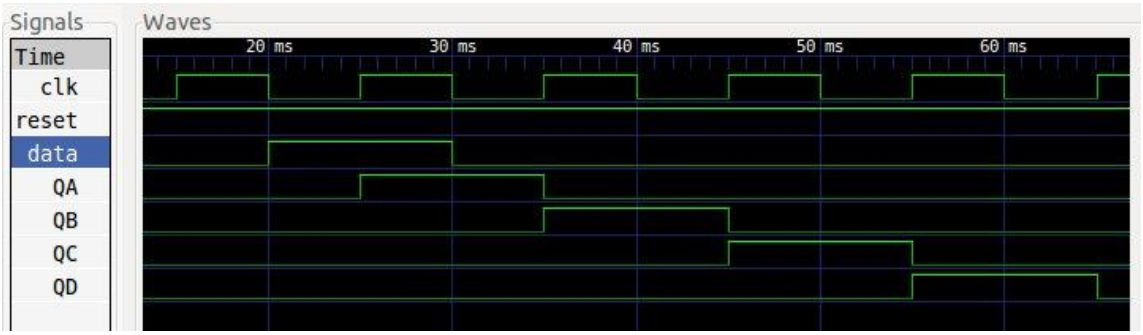


Obrázek 15 - Schéma posuvného registru.

Způsob řešení posuvného registru bude shodný s předchozí úlohou, a to jak pomocí editoru schémat, následně dvěma zápisy pomocí jazyka Verilog. Soubory budou obdobně označeny, např. pro editor schémat *PosuvReg_sch.v.* Jednotlivá řešení jsou opět otestována pomocí simulace a následně zobrazena v programu GTKWawe.

Pořadí jednotlivých příkazů volaných v příkazové řádce:

- zkompilujeme jednotlivé moduly příkazem *iverilog -o reg PosuvReg_tb.v PosuvReg_sch.v PosuvReg_verilog1.v XOR_verilog2.v*,
- provedeme simulaci (textový výpis) příkazem vvp reg,
- jednotlivé průběhy ze simulace graficky porovnáme v programu GTKWave příkazem *gtkwave test.vcd &*.



Obrázek 16 - Zobrazení průběhu simulace v programu GTKWave.

4.4.3 DETEKCE VZESTUPNÉ A SESTUPNÉ HRANY

Detekce hran signálů patří mezi základ návrhu číslicových obvodů. Podle typu hrany může systém patřičně reagovat. Pro vysvětlení principu byl vytvořen modul, který detekuje požadovaný typ hrany, dáno klausulí posedge, negedge. V případě, že není klausule uvedena, systém reaguje na obě tyto hrany.

Obrázek 17 - Detekce hran signálů.

```
module hrana(
input wire hrana_p,     // vstup pro vzestupnou hranu
input wire hrana_n,     // vstup pro sestupnou hranu
input wire reset,       // vstup reset
output reg out_a,       // vystup A
output reg out_b        // vystup B
);

always @(posedge reset or posedge hrana_p or
negedge hrana_n)
begin
 if (reset)
   begin
     out_a <= 1'b0 ;
     out_b <= 1'b0 ;
   end
  else
     begin
       if (hrana_p)
          out_a <= 1'b1 ;
       else
          out_b <= 1'b1 ;
     end
end
endmodule
```

Obrázek 18 - Ukázka Verilog kódu.

4.4.4 SYNCHRONNÍ A ASYNCHRONNÍ RESET

Rozdíl mezi synchronním a asynchronním resetem je popsán na modifikovaném binární čítači. Čítač obsahuje oba tyto typy resetů, ten asynchronní je uveden v definici bloku *always*. Provede se vždy při náběžné hraně signálu (dáno klausulí *posedge*), není tedy nijak svázán s hodinovým signálem. Oproti tomu synchronní reset je svázán s hodinovým signálem, tj. provede vždy až při následující vzestupné hraně hodinového signálu, viz obrázek č. 19.

Obrázek 19 - Synchronní a asynchronní reset.

```
module binary_counter2    (
input wire clk,          // vstup hodinoveho signalu
input wire s_reset,   // vstup synchronni reset citace
input wire a_reset,   // vstup asynchronni reset citace
output reg [7:0] out  // vystup citace
);

always @(posedge clk or posedge a_reset )
begin
 if (a_reset)
    out <= 8'b0 ;
  else
    begin
      if (s_reset)
          out <= 8'b0 ;
      else
          out <= out + 1;
    end
end
endmodule
```

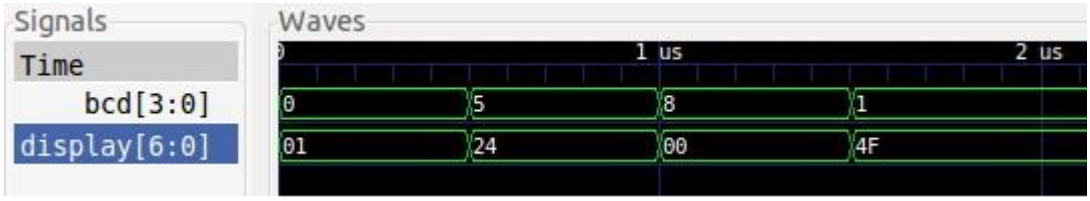
Obrázek 20 - Ukázka Verilog kódu.

4.4.5 DEKODÉR KÓDU BCD NA 7-SEGMENT

Binárně zakódované desítkové číslo se převede na sedmisegmentový display pomocí modulu *dekodér* využívající příkaz *case*. Ukázka zobrazuje shodnost syntaxe jazyka Verilog s jazykem C. Podle hodnoty vstupu *bcd* je v dekodéru přiřazena hodnota výstupu. Dekodér reaguje ihned na změnu vstupu, tj. není zde potřeba žádná synchronizace popř. pomocný hodinový signál.

```
module bcd7 (input [3:0] bcd, output reg [6:0] display);
always @* begin
  case(bcd)
  4'b0000: display = 7'b1111110;
  4'b0001: display = 7'b0110000;
  4'b0010: display = 7'b1101101;
  4'b0011: display = 7'b1111001;
  4'b0100: display = 7'b0110011;
  4'b0101: display = 7'b1011011;
  4'b0110: display = 7'b1011111;
  4'b0111: display = 7'b1110000;
  4'b1000: display = 7'b1111111;
  4'b1001: display = 7'b1111011;
  default: display = 7'b0000000;
  endcase
  display = ~display;
end
endmodule
```

Obrázek 21 - Ukázka Verilog kódu.



Obrázek 22 - Ukázka simulace BCD dekodéru.

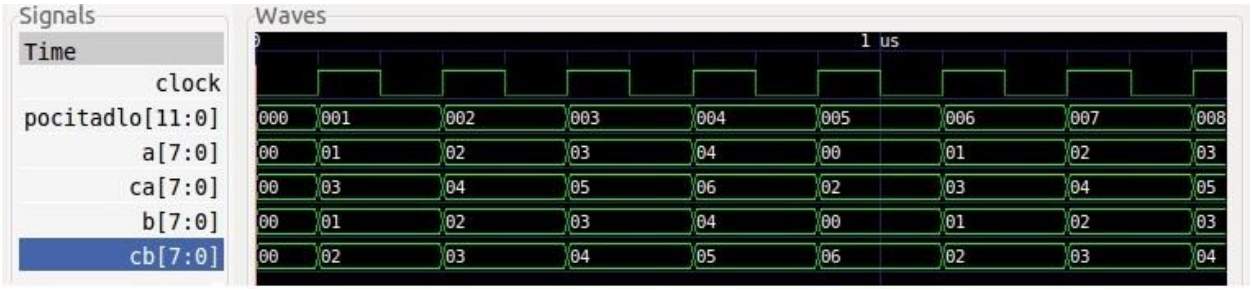
4.4.6 BLOKUJÍCÍ A NEBLOKUJÍCÍ PŘIŘAZENÍ

Kapitola popisuje rozdíl mezi blokujícím a neblokujícím přiřazením na praktickém příkladě. Modul obsahuje počitadlo hodinových impulzů a dvě proměnné a a b. První proměnná a používá neblokující přiřazení (značí se „=“) a druhá b blokující přiřazení (značí se „<=“). Aby pomocí simulace byl patrný rozdíl mezi přiřazením, modul obsahuje další dvě pomocné proměnné ca, cb, které přičítají k daným proměnným číslo dvě.

```
module blokujici( input clk);

reg [11:0] pocitadlo=0;    //pocitadlo hodinovych impulzu
reg [7:0] a=0;   reg [7:0] ca=0;
reg [7:0] b=0;   reg [7:0] cb=0;

always @(posedge clk )
begin
   pocitadlo=pocitadlo+1;
   a  =  (pocitadlo==5) ? 0: a+1;
   b <= (pocitadlo==5) ? 0: b+1;
  ca= a+2;
  cb= b+2;

end
endmodule
```

Obrázek 23 -Ukázka Verilog kódu.

Obrázek č. 24 znázorňuje průběh hodnot všech proměnných během simulace. Z tohoto obrázku je na první pohled patrný rozdíl. U blokujícího přiřazení se změněná hodnota proměnné projeví až při dalším cyklu, tj. až při další vzestupné hraně. Kdežto u neblokujícího přiřazení se hodnota změní ihned a další výpočty už pracují s touto pozměněnou hodnotou. Proto programátor musí vždy zvážit, kde jaké přiřazení je tou nejlepší volbou.



Obrázek 24 - Ukázka průběhu simulace.

5 PYTHON

Python je dynamický objektově orientovaný skriptovací programovací jazyk, který v roce 1991 navrhl Guido van Rossum. Python je vyvíjen jako open source projekt, který zdarma nabízí instalační balíky pro většinu běžných platforem (Unix, Windows, Mac OS), ve většině distribucí systému Linux je Python součástí základní instalace.

Základy programovacího jazyka jsou podrobně popsány ve zdroji [1] a nejsou cílem této práce. V následujících kapitolách bude popsána základní tvorba programu v jazyce Python se zaměřením na tvorbu modulů potřebných pro komunikaci s modulem DE0-Nano.

Obrázek 25 - Logo skriptovacího jazyka Python. Zdroj [6].

5.1 PYTHON V PŘÍKAZOVÉM ŘÁDKU

Nejjednodušší způsob seznámení s uvedeným jazykem je spuštění jeho interpreteru v terminálu příkazem *python*, po kterém se objeví prompt ve tvaru *„>>>“* (tzv. primární výzva). Po této výzvě je pak schopen přijímat naše příkazy, např. *„5+3“*. Každý příkaz se provede po stisku klávesy *Enter*.

Interpreter pracuje podobně jako shell systému Linux, znázorněno na ukázce výpisu hodnot promněné *x*, viz obrázek č. 26. Po stisku klávesy *Enter* se zobrazí *„...“*, tzv. sekundární výzva. Ta slouží pro zadávání složitějších příkazů rozložených do více řádků, v našem případě opět stiskneme klávesu *Enter*. Pro ukončení činnosti ve zmíněném prostředí stačí zadat příkaz *quit()* a vrátíme se zpět do terminálu.

```
for x in 1, 2, 3, 4, 5: print x
```

Obrázek 26 - Ukázka zdrojového kódu v jazyce Python.

5.2 PYTHON SKRIPT

Jak již bylo zmíněno, jazyk Python je skriptovací jazyk a v této podobě se taky z důvodu jednoduchosti a rychlosti psaní programů často používá. Zde se zaměříme na tvorbu jednoduchého skriptu posílajícího přes rozhraní USB port řetězec znaků v prostředí OS Linux.

V první fázi zjistíme pomocí příkazu *dmesg | grep tty*, na jakém USB portu se naše zařízení nachází. Následuje vytvoření souboru python1.py, např. pomocí Midnight Commanderu (klávesy Shift+F4). Zdrojový kód je zobrazen v obrázku č. 27. Pro komunikaci přes USB či seriové porty musí být nainstalován balíček *pyserial*. Instalace balíčku se provede příkazem *pip install pyserial*. Pip je package *manager* pro Python a slouží k instalaci, odebírání a aktualizaci všech knihoven. Pokud není správce balíčků pip nainstalován, doinstaluje se příkazem *sudo apt-get install python-pip*.

```
#!/usr/bin/env python
import serial

ser = serial.Serial('/dev/ttyUSB0', 9600)
ser.write("DE0-Nano\n")
ser.close()
```

Obrázek 27 - Ukázka zdrojového kódu. Soubor: python1.py

Uvedený skript spustíme příkazem *python python1.py*. Na BSD kompatibilních systémech lze libovolný skript jazyka Python změnit na spustitelný soubor uvedením "magické" sekvence „*#! /usr/bin/env python*“ na začátku souboru (obdobný zápis se píše ve skriptech unixového shellu) a nastavením potřebných práv k souboru (např. *chmod a+x *.py)*.

5.3 GUI POMOCÍ KNIHOVNY PYQT

PyQt je knihovna pro integraci GUI knihovny Qt do Pythonu. Alternativní řešení pro tvorbu GUI nabízí také PySide (Qt vazba s oficiální podporou a volnější licencí), PyGTK, wxPython a Tkinter (který je dodáván s Pythonem). PyQt je vyvíjen Britskou firmou Riverbank Computing. Je dostupný kromě komerční verze také pod licencí GNU/GPL-2 pro operační systémy Linux, Unix, Mac OS X a Microsoft Windows. Instalace knihovny, pokud už není součástí distribuce, se provede příkazem *sudo apt-get install python-qt4*.

Ukázka nejjednodušší grafické aplikace je zobrazena na obrázku č. 28. Tato ukázka se skládá z importu grafické knihovny PyQT a vytvoření objektu aplikace *app* s jedním widgetem *wid*.

```
#! /usr/bin/env python
import sys
from PyQt4.QtGui import *

app = QApplication(sys.argv)
wid = QWidget()
wid.resize(250, 150)
wid.move(100,100)
wid.setWindowTitle("Aplikace v Pythonu")
wid.show()

sys.exit(app.exec_())
```

Obrázek 28 - Ukázka zdrojového kódu. Soubor: python2.py

Pokud upravíme aplikaci tak, aby se chovala shodně s aplikací uvedenou v kapitole 5.2, vložíme do widgetu tlačítko, které po stisku vyšle požadovaný text. Metoda *connect* propojí signál *clicked*, který je vygenerovaný objektem *QPushButton* se slotem *onclickSlot()*.

```
#! /usr/bin/env python
import serial
import sys
from PyQt4.QtGui import *

def onclickSlot():
    ser = serial.Serial('/dev/ttyUSB0', 9600)
    ser.write("DE0-Nano\n")
    ser.close()

ap = QApplication(sys.argv)
wid = QWidget()
wid.resize(250, 150)
wid.move(100,100)
wid.setWindowTitle("Aplikace v Pythonu")

btn = QPushButton('Odesli', wid)
btn.setToolTip('Odesle text na USB')
btn.resize(btn.sizeHint())
btn.move(80, 50)
btn.clicked.connect(onclickSlot)

wid.show()
sys.exit(ap.exec_())
```

Obrázek 29 - Ukázka zdrojového kódu. Soubor: python3.py

Předchozí ukázky programů byly napsány procedurálním stylem za použití moderního OOP lze stejný program napsat takto:

```
class MojeApp(QtGui.QWidget):
    def __init__(self, parent=None):
        super(MojeApp, self).__init__(parent)

        self.ser=0
        self.openButton = QtGui.QPushButton("&Open port")
        self.openButton.show()
        self.sendButton = QtGui.QPushButton("&Send data")
        self.sendButton.show()
        self.closeButton = QtGui.QPushButton("&Close port")
        self.closeButton.show()
        self.closeButton.setEnabled(False)
        self.sendButton.setEnabled(False)

        self.openButton.clicked.connect(self.openPort)
        self.sendButton.clicked.connect(self.sendData)
        self.closeButton.clicked.connect(self.closePort)

        mainLayout = QtGui.QVBoxLayout()
        mainLayout.addWidget(self.openButton)
        mainLayout.addWidget(self.sendButton)
        mainLayout.addWidget(self.closeButton)
        mainLayout.addStretch()

        self.setLayout(mainLayout)
        self.setWindowTitle("Aplikace v Pythonu")
        self.setFixedSize(250,100)
        self.move(100,100)

    def openPort(self):
        self.openButton.setEnabled(False)
        self.closeButton.setEnabled(True)
        self.sendButton.setEnabled(True)
        self.ser = serial.Serial('/dev/ttyUSB0', 9600)

    def closePort(self):
        self.openButton.setEnabled(True)
        self.closeButton.setEnabled(False)
        self.sendButton.setEnabled(False)
        self.ser.close()

    def sendData(self):
        self.ser.write("DE0-Nano\n")


if __name__ == '__main__':
    import sys

    app = QtGui.QApplication(sys.argv)

    mojeapp = MojeApp()
    mojeapp.show()

    sys.exit(app.exec_())
```

Obrázek 30 - Ukázka zdrojového kódu. Soubor: python4.py.

Moduly v Pythonu jsou objekty a všechny mají zabudovaný atribut __name__. Jeho hodnota závisí na tom, jakým způsobem je modul využit, např. pokud modul je spuštěn jako samostatný program (jako v ukázce), bude obsahovat hodnotu __main__. V tomto případě budou vytvořeny objekty *app*, *mojeapp* a aplikace bude spuštěna.

5.4 GUI POMOCÍ QT DESIGNERU

Další z možností jak vytvořit grafickou aplikaci v Pythonu, je využití QT Designeru. Ten nainstalujeme pomocí příkazu *sudo apt-get install qt4-designer*. V designeru vytvoříme požadovaný vzhled grafické aplikace. Popis grafického rozhraní je uložen do souboru s příponou ui ve formátu XML, viz ukázka kódu.

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>Dialog</class>
<widget class="QDialog" name="Dialog">
<property name="geometry">
  <rect>
    <x>0</x>
    <y>0</y>
    <width>300</width>
    <height>230</height>
  </rect>
</property>
```

Obrázek 31 - Ukázka části zdrojového kódu. Soubor: GUIdialog.ui.

Tento soubor převedeme do kódu pro Python pomocí utility *pyui4*, která je součástí balíčku *Development tools for PyQt4*, který nainstalujeme příkazem *sudo apt-get install pyqt4-dev-tools.*

```
pyuic4 -o python5_gen.py -x GUIdialog.ui
```

Obrázek 32 - Ukázka linuxového příkazu pro převod z PyQt4.

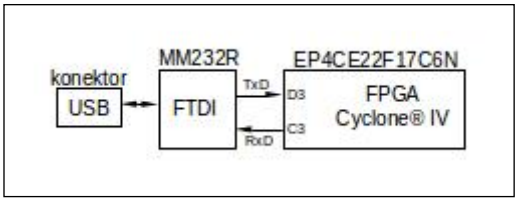
Tento soubor přejmenujeme na python5.py a zdrojový kód doplníme o požadovanou reakci na stisk jednotlivých tlačítek. Zde je použit starší typ propojení signál-slot využívající metodu *QtCore.QObject.connect*.

```
class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName(_fromUtf8("Dialog"))
        Dialog.resize(300, 230)
        ...
        self.buttonBox = QtGui.QDialogButtonBox(Dialog)
        self.buttonBox.setGeometry(QtCore.QRect(-90, 180, 341, 32))
        self.buttonBox.setOrientation(QtCore.Qt.Horizontal)
        self.buttonBox.setStandardButtons(QtGui.QDialogButtonBox.Cancel|QtGui.QDialogButtonBox.Ok)
        self.buttonBox.setObjectName(_fromUtf8("buttonBox"))
        self.groupBox = QtGui.QGroupBox(Dialog)
        self.groupBox.setGeometry(QtCore.QRect(30, 20, 251, 151))
        self.groupBox.setAutoFillBackground(False)
        ...
        self.retranslateUi(Dialog)
        …
        QtCore.QObject.connect(self.pushButtonOpen, QtCore.SIGNAL(_fromUtf8("clicked()")), self.openPort)
        QtCore.QObject.connect(self.pushButtonClose, QtCore.SIGNAL(_fromUtf8("clicked()")), self.closePort)
        QtCore.QObject.connect(self.pushButtonSend, QtCore.SIGNAL(_fromUtf8("clicked()")), self.sendData)
```

Obrázek 33 - Část zdrojového kódu - propojení signál-slot. Soubor: python5.py.

5.5 OVLÁDÁNÍ OBVODU FTDI POMOCÍ SKRIPTŮ V PYTHON

Obvod FT232R je převodník USB/UART disponující přenosovou rychlostí až 3 MB/s. Součástí obvodu je interní 1kB EEPROM pro uložení uživatelských dat. Komunikace s FPGA probíhá pomocí dvou signálů RXD, TXD, signály pro hardwarové řízení komunikace (RTS, CTS, DTS, DTR, DSR, DCD) nejsou využity. Obvod lze napájet i ze sběrnice USB. Pro účely vývoje a testování byl použit modul MM232R, osazený zmíněným obvodem, který lze využívat i při práci s nepájivým polem.



Obrázek 34 - Ukázka propojení FPGA s obvodem FTDI.

Pro komunikaci s obvodem FTDI je nutné nainstalovat základní knihovnu *libftdi-dev* příkazem *sudo apt-get install libftdi-dev*. Dále pak knihovnu *pylibftdi*, která usnadní komunikaci s daným obvodem. Instalace se provede příkazem *pip install pylibftdi*. Tato knihovna obsahuje i modul pro vyhledávání připojených FTDI obvodů, ve výpise zobrazí typ a identifikace obvodu, které zadáváme při vytváření instance zařízení. Příkaz voláme *sudo python -m pylibftdi.examples.list_devices*. Parametr *-m* nám umožňuje spouštět moduly uložené v sys.path jako skripty. Adresáře uložené v sys.path lze zobrazit následujícím skriptem, viz obrázek č. 35.

```
#!/usr/bin/env python
import sys
from pprint import pprint as p

p(sys.path)
```

Obrázek 35 - Ukázka zdrojového kódu. Soubor: python6.py.

Dočasné přidání nové cesty do sys.path řeší skript python7.py, který doplní seznam o adresář /home/user/python. V rámci této session jsou pak dostupné moduly v daném adresáři.

```
#!/usr/bin/env python
import sys
import os
from pprint import pprint as p

path = "/home/cisco/python"

if not os.path.isdir(path):
    os.mkdir( path, 0755 )
sys.path.append(path)

p(sys.path)
```

Obrázek 36 - Ukázka části zdrojového kódu. Soubor: python7.py.

Pokud požadujeme trvalé přidání adresáře, modifikujeme cestu PYTHONPATH přidáním příkazu *export PYTHONPATH=/home/cisco/python:$PYTHONPATH* na konec souboru *~/.bash.rc*. Změna se projeví až po spuštění nového terminálu.

Komunikaci s obvodem FT232R lze rozdělit do dvou částí, a to standardní sériová komunikace, viz obrázek č. 37, nebo režim bit-bang, viz obrázek č. 38. V tomto režimu lze u obvodu ovládat samostatně jednotlivé piny, což umožňuje v jazyce Python generovat požadované průběhy signálů. Tento způsob lze využít pro otestování funkčnosti zařízení, popř. k otestování Verilog modulu.

```
#!/usr/bin/env python
from pylibftdi import Device

with Device(device_id='FTDHNZSD',mode='t') as dev:
        dev.baudrate = 2400
        dev.write('Posilam text')
```

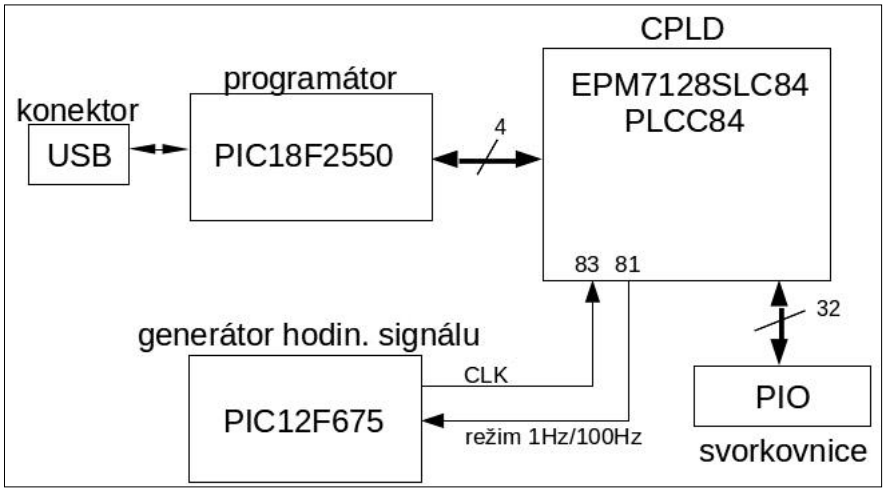
Obrázek 37 - Ukázka zdrojového kódu. Soubor: python8.py.

```
#!/usr/bin/env python
import time
from pylibftdi import BitBangDevice


dev=BitBangDevice(device_id='FTDHNZSD')
dev.direction = 0x01  # 0 bit vystup, ostatni
vstupy

dev.port &= 0xFE      # clear bit 0
time.sleep(1)
dev.port |= 1         # set bit 0
time.sleep(1)
dev.port &= 0xFE      # clear bit 0
time.sleep(1)
dev.port |= 1         # set bit 0
```

Obrázek 38 - Ukázka zdrojového kódu. Soubor: python9.py.

6 VÝVOJOVÝ CPLD KIT

Pro prvotní účely testování jazyka Verilog byl navržen vývojový kit s obvodem CPLD (Complex Programmable Logic Devices) EPM7128SLC84 od firmy Altera. Pro snadnou výrobu byl realizován na jednostranné desce za použití návrhového softwaru Eagle PCB. Deska má rozměry 120 x 80 mm a je celá napájena prostřednictvím USB portu.

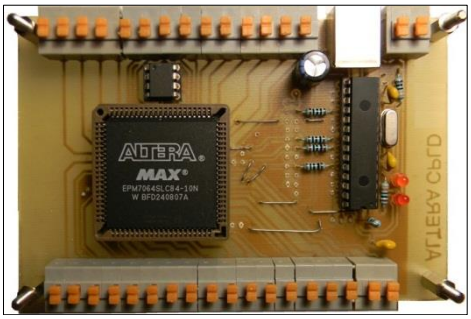


Obrázek 39 - Blokové schéma CPLD kitu

Parametry CPLD EPM7128SLC84:

- obvod řady MAX 7000,
- pouzdro PLCC84,
- vysoce výkonné EEPROM programovatelné logické pole (PLD),
- 5V logika,
- vestavěné JTAG rozhraní,
- obsahuje 128 makrobuněk,
- zpoždění 5 ns pin-to-pin,
- maximální frekvence až 175,4 MHz.

Pro konstrukci programátoru bylo použito zapojení dle [13] s procesorem PIC18F2550, které je kompatibilní s ByteBlaster programátorem a lze tedy desku programovat přímo z vývojového prostředí Quartus II. Nutno však použít starší verzi 13.0, neboť v novějších verzích už tato řada CPLD není podporována.



Obrázek 40 - Fotografie CPLD kitu

Jako zdroj hodinového signálu, z důvodu požadované nízké frekvence a minimálního počtu potřebných součástek, byl použit mikroprocesor PIC12F675. Ten pracuje ve dvou režimech, a to 1 Hz a 100 Hz podle úrovně na vstupu GP1. Důvod nízké pracovní frekvence vychází z potřeb zobrazovat výstupy testovaných modulů prostřednictvím LED diod, tedy bez potřeby digitálního osciloskopu. Výstup hodinového signálu na pinu GP2 se střídou 50 : 50 je generován pomocí přerušení od časovače Timer0.

```
//interrupt obsluha
void interrupt isr(void) {
   static uint32_t count=0; // pocitadlo
   uint16_t pom;

   TMR0 += 256 - 250 + 3; // nastav hodnotu pro Timer0
   INTCONbits.T0IF = 0; // smaz interrupt flag
   ++count; // prirustek kazdych 250 us

   //volba frekvence
   my_bits.in_volba= GPIObits.GP1;


   if (my_bits.in_volba == 1) pom = 500000 / 250; // kazdych 500 ms, strida 50:50
     else pom = 5000 / 250; // kazdych 5 ms, strida 50:50

   if (count >= pom) {
      count = 0;
      my_bits.out_clk = ~my_bits.out_clk;
      GPIObits.GP2 = my_bits.out_clk;

   }
}
```

Obrázek 41 - Část zdrojového kódu pro PIC12F675

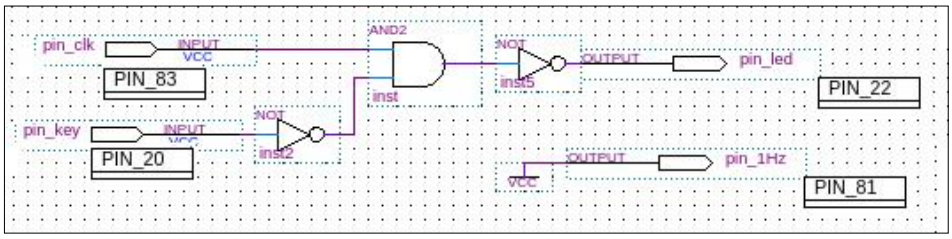
Kit je opatřen pružinovou svorkovnicí ARK124A obsahující 32 pinů, které lze naprogramovat jako vstupní, výstupní nebo obousměrné. K této svorkovnici lze připojit tlačítka, LED diody a ostatní potřebné obvody. Všechny piny jsou s 5V logikou (dáno typem CPLD) a nemají žádnou přepěťovou ochranu ani ochranu proti zkratu. Absence ochrany obvodu CPLD značně zjednodušila návrh desky a v případě důsledné kontroly připojených obvodů je i zbytečná.

svorkovnice	pin CPLD	svorkovnice	pin CPLD	svorkovnice	pin CPLD
pin1	20	pin12	51	pin23	9
pin2	22	pin13	54	pin24	5
pin3	28	pin14	56	pin25	79
pin4	30	pin15	58	pin26	80
pin5	33	pin16	60	pin27	77
pin6	35	pin17	63	pin28	75
pin7	37	pin18	64	pin29	76
pin8	39	pin19	18	pin30	74
pin9	41	pin20	16	pin31	70
pin10	45	pin21	12	pin32	68
pin11	49	pin22	11		

Obrázek 42 - Tabulka připojení pinů CPLD na svorkovnici.

6.1 PRVNÍ ÚLOHA – BLIKAČ S LED

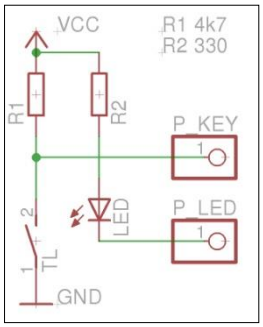
Cílem úlohy je vytvořit blikač s LED diodou, který bude aktivní při stisku tlačítka. Frekvenci blikání zvolíme 1 Hz, proto úroveň na vstupu generátoru hodinového signálu GP1 bude logická jednička. Pro návrh bude využit editor schémat. Blokové schéma návrhu je zobrazeno na obrázku č. 43.



Obrázek 43 - Blokové schéma.

Zde se využívá vlastnost hradla AND a úloha slouží pro prvotní seznámení s obvodem CPLD. Po spuštení vývojového prostředí Quartus zadáme volbu vytvořit nový projekt. Zde zadáme typ našeho CPLD obvodu. Dále následuje vytvoření nového souboru pro nakreslení požadovaného schematu (sch file). U v/v pinů musíme provést propojení jednotlivých vývodů k fyzickým pinům CPLD obvodu (Pin Planner), pro obvod generátoru hodinového signálu jsou piny pevně dány.

Na výstupu z hradla AND je umístěno hradlo negace z důvodu, aby v klidovém stavu LED dioda nesvítila. Ze znalosti logických hradel se také nabízí možnost použití hradla NAND. Následuje kompilace projektu a naprogramování CPLD. K otestování úlohy stačí připojit ke kitu LED dioda a tlačítko, viz obrázek č. 44.



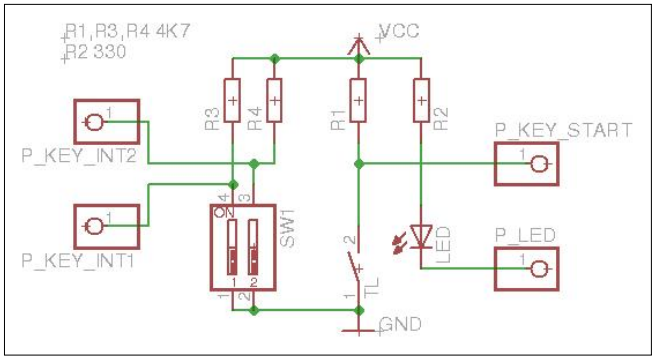
Obrázek 44 - Ukázka zapojení součástek.

6.2 DRUHÁ ÚLOHA – PROGRAMOVATELNÝ MKO

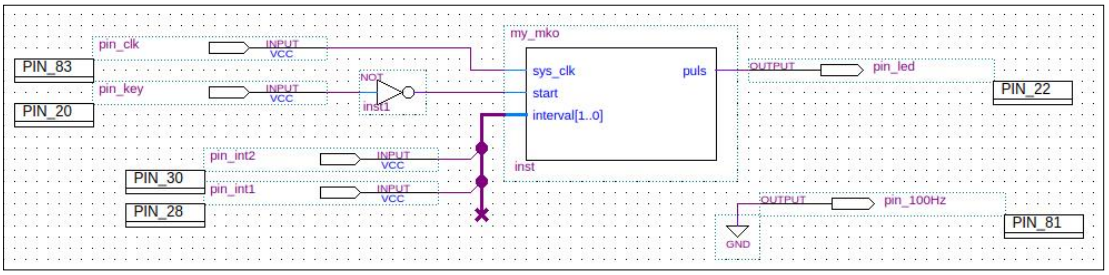
Druhá úloha bude realizovat programovatelný monostabilní obvod, obdobu NE555. Délka intervalu výstupního signálu však nebude dána kombinací RC, ale kombinací dvou vstupních bitů. Pomocí těchto bitů budeme moci měnit šířku ve 4 krocích. Pro usnadnění je zde vytvořen skript v jazyce Python, který vygeneruje potřebný modul v jazyce Verilog z požadovaných parametrů. Skript gen_mko.py se volá s následujícími parametry:

- název generovaného modulu, zadáno bez přípony.v,
- vstupní frekvence hodinového signálu (Hz),
- délka výstupního signálu pro kombinace vstupních bitů 00 (µs),
- délka výstupního signálu pro kombinace vstupních bitů 01 (µs),
- délka výstupního signálu pro kombinace vstupních bitů 10 (µs),
- délka výstupního signálu pro kombinace vstupních bitů 11 (µs).

Vygenerovaný Verilog soubor pak importujeme do projektu v prostředí Quartus. Následuje vytvoření schématické značky z tohoto modulu a vložení do schématu. Po kompilaci projektu naprogramujeme CPLD a projekt otestujeme pomocí obvodu zobrazeném na obrázku č. 45.



Obrázek 45 - Ukázka zapojení součástek.



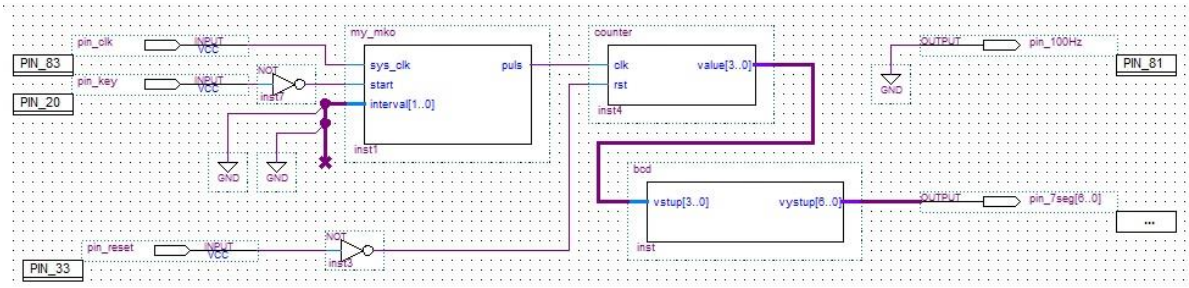
Obrázek 46 - Ukázka MKO.

Výstupní generovaný signál je synchronizován se vzestupnou hranou hodinového signálu, z důvodu přesného generování výstupního impulsu. Je nutno však počítat s tím, že výpočet hodnot vnitřního čítače se provádí jako podíl délky impulzu ke vstupní frekvenci a výsledná hodnota výpočtu je zaokrouhlena na celé číslo, což nejvíce ovlivňuje výslednou přesnost.

Generátor hodinového signálu s PIC je přepnut do režimu 100 Hz pomocí vstupu GP1, který je nastaven na logickou nulu. Startovací signál je přiveden na PIO0, výstup MKO je na PIO1, na který je připojena LED dioda. Podobným způsobem můžeme realizovat PWM, např. pro řízení jasu LED diody.

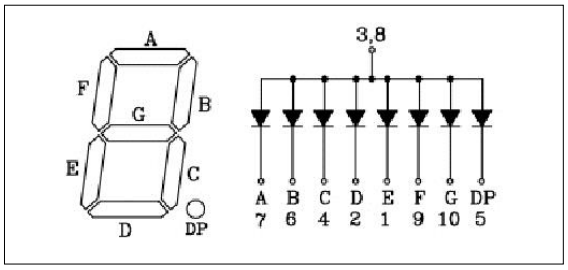
6.3 TŘETÍ ÚLOHA – ČÍTAČ IMPULSŮ SE SEDMISEGMENTOVKOU

Tato úloha bude realizovat čítač impulsů, kdy po stisknutí tlačítka bude inkrementována hodnota na displeji. V ukázce bude použit předchozí modul MKO pro ošetření případných zákmitů tlačítka. Dále zde bude pomocí jazyka Verilog vytvořen dekodér kódu BCD na 7-segment. Pro návrh bude opět využit editor schémat. Blokové schéma a schéma návrhu je zobrazeno na obrázku č. 47.



Obrázek 47 - Ukázka zapojení čítače.

Převodní tabulka dekodéru BCD na 7-segment je tvořena příkazem *case*, kde jako vstupní parametr je hodnota binárního čítače. Ta se pohybuje v rozmezí 0 až 9, poté je nutno čítač opět resetovat. Výstup z dekodéru je přiveden přes rezistory na LED sedmisegmentovku, označení jednotlivých segmentů je zobrazeno na obrázku č. 48.



Obrázek 48 - Vnitřní zapojení sedmisegmentovky.

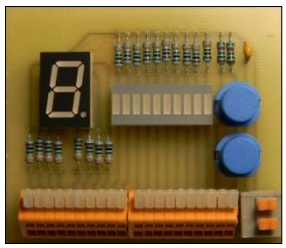
Úloha ukazuje možnost kombinace Verilog kódu se schématickým návrhem, zde vývojové prostředí umožňuje z libovolného modulu vytvořit schématickou značku. Toto řešení do jisté míry zpřehledňuje a zjednodušuje celý návrh obvodu.

```
module bcd( input [3:0] vstup, output reg [6:0]
vystup);

always @ (vstup)
begin
  case (vstup)  // abcdefg
    0 : vystup = 7'b0000001;
    1 : vystup = 7'b1001111;
    2 : vystup = 7'b0010010;
    3 : vystup = 7'b0000110;
    4 : vystup = 7'b1001100;
    5 : vystup = 7'b0100100;
    6 : vystup = 7'b0100000;
    7 : vystup = 7'b0001111;
    8 : vystup = 7'b0000000;
    9 : vystup = 7'b0000100;
    default : vystup = 1;
  endcase
end

endmodule
```

Obrázek 49 - Ukázka zdrojového kódu.

6.4 DALŠÍ VYUŽITÍ CPLD KITU

Na modulu byly také realizovány úlohy popisované v kapitole Icarus Verilog. Pro snadnější testování verilog modulů byl navržen přípravek obsahující 2 tlačítka, 8 LED diod a jednu sedmisegmentovku.



Obrázek 50 - Fotografie přípravku.

Pomocí tohoto kitu, který byl rozšířen o převodník USB-FIFO, bylo také realizováno řízení robotické ruky. Na čipu CPLD byly vytvořeny registry a jednotlivé PWM bloky pro řízení modelářských serv HITEC.

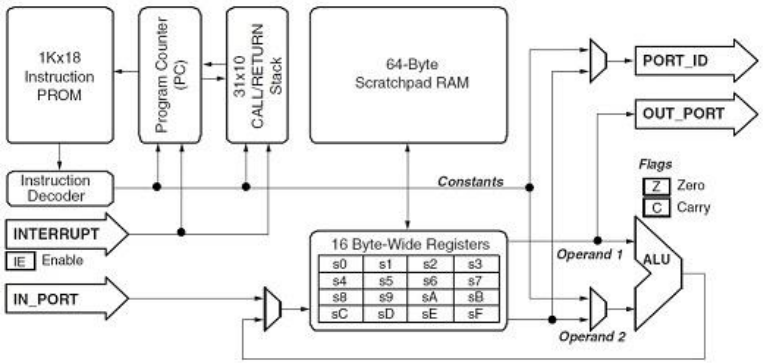
7 NÁVRH 8-BITOVÉHO SOFT-PROCESORU

Kapitola popisuje vytvoření jednoduchého 8-bitového soft-procesoru na FPGA. Návrh vychází ze znalostí vnitřních architektur mikroprocesorů ATMEL a PIC. Součástí návrhu je vlastní kompilátor assembleru napsaný v jazyce Python. Zmíněný soft-procesor byl z důvodu snadného pochopení principu složen z běžně známých funkčních bloků s absencí jakékoliv optimalizace. Každý funkční blok obsahuje soubor testbench pro otestování jeho funkčnosti. V závěru je otestována funkčnost celého řešení na testovacích příkladech. Pro realizaci soft procesoru je nutné mít na PC nainstalované následující programy: Icarus Verilog, GTKWave, Quartus, Python.

7.1 PROCESOROVÉ JEDNOTKY

Procesorové jednotky se dělí na dvě velké skupiny. První skupinou jsou tzv. soft-procesory vytvořené pomocí základních stavebních bloků na hradlovém poli. Výhodou soft struktur je snadná konfigurace a přenositelnost mezi jednotlivými řadami hradlových polí. Řady soft-procesorových jader jsou velmi rozmanité od 8bitových procesorů, zabírajících několik desítek slices, až po výkonné 64bitové systémy vyžadující desetitísíce slices. Obvykle jsou soft-jádra dodávána výrobci hradlových polí (firma Xilinx – 8bitový procesor Picoblaze, 32bitový Microblaze, od firmy Altera je známý 32bitový procesor Nios, firma Lattice nabízí procesor LatticeMicro32). Nevýhodou soft řešení je nižší pracovní frekvence (např. již zmíněné low cost řešení dosahuje pracovní frekvence u 32bitového mikroprocesoru kolem 66 MHz).

Druhou skupinou jsou jádra integrovaná přímo do křemíku. V těchto případech je velmi omezená konfigurace procesoru, ovšem procesory pracují kolem frekvence 1 GHz s podporou jednotek správy paměti MMU. Dnešní trend je integrace vícejádrových procesorů typu ARM do FPGA.

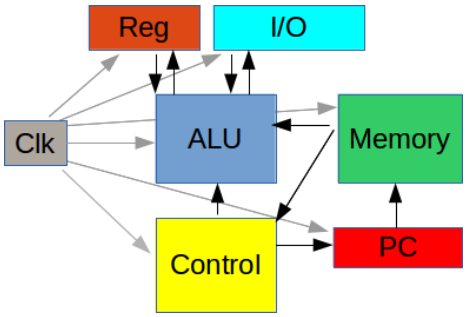


Obrázek 51 - Soft-procesor Picoblaze. Zdroj [9].

7.2 ARCHITEKTURA SOFT PROCESORU

Architektura procesoru vychází z Harvardské architektury a je zobrazena na obrázku č. 52. Skládá se z následujících funkčních bloků, které plní tyto funkce:

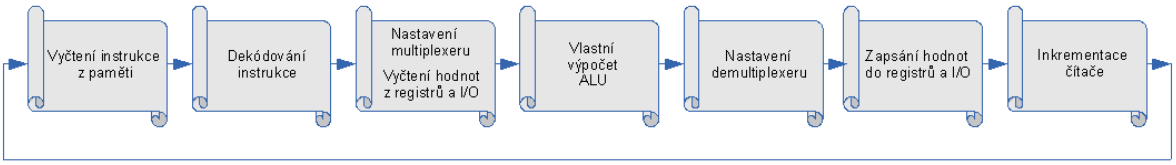
- ALU - aritmetické a logické operace,
- Memory - obsahuje přeložený program,
- PC - počítadlo programu včetně řízení skoků,
- Reg - registry procesoru včetně registru příznaků,
- I/O - vstupně/výstupní porty,
- Clk - modul časování procesoru,
- Control - modul řízení procesoru.



Obrázek 52 - Architektura soft-procesoru.

Každá instrukce potřebuje pro zpracování 7 hodinových impulsů a je prováděna samostatně, nedochází k překrývání instrukcí, tzv. pipelining. U mikroprocesorů Microchip se využívají 4 hodinové impulsy a u historické 8051 jich bylo 12. V případě soft-procesoru je to dáno postupným zpracováním informace v několika samostatných funkčních blocích. Řešení není sice optimální, ale zjednodušuje vlastní návrh procesoru a umožňuje jeho simulaci a zobrazení průběhů v programu GTKWave. Časová posloupnost zpracování

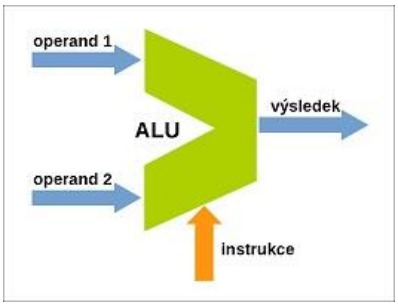
instrukce je zobrazena na obrázku č. 53. Modul *clock_pipe.v* z tohoto důvodu generuje samostatné časově posunuté hodinové impulsy pro jednotlivé bloky, obdoba posuvného registru.



Obrázek 53 - Časová posloupnost zpracování instrukce.

7.3 POPIS POUŽITÝCH MODULŮ

Aritmeticko-logická jednotka (ALU arithmetic logic unit) je základní komponentou soft-procesoru. Jednotka provádí základní aritmetické (součet, rozdíl) a logické operace (logický součin, logický součet, negaci, rotaci). Každou náběžnou hranou hodin tohoto modulu je provedena zvolená operace se vstupními operandy. Hodinový signál je získán z modulu clock_pipe, který zajišťuje synchronizaci provádění výpočtu (platné hodnoty operandů).



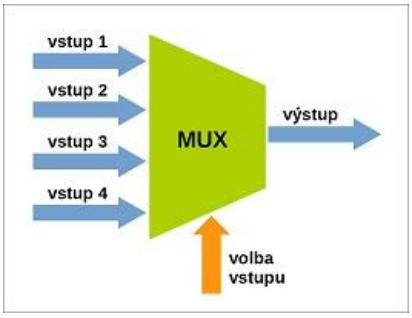
Obrázek 54 - Zjednodušené schéma ALU.

Řadič je druhou nejdůležitější komponentou a slouží k dekódování instrukce, určení adresní části instrukce a jeho operandů. Tato komponenta je zodpovědná za zpracování tří instrukcí RESET, JMP a JZ.

Jednotka hodinového signálu (*clock_pipe.v*) slouží k synchronizaci chodu celého soft-procesoru. Je zodpovědná za generování jednotlivých hodinových signálů pro moduly a zajišťuje zpracování instrukce v 7 krocích, viz obrázek č. 53.

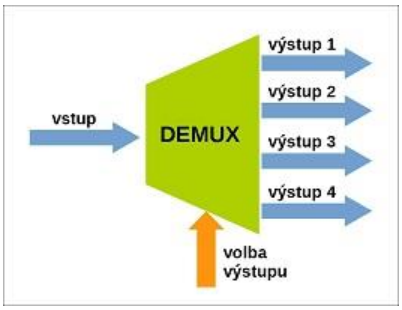
Registr je nejjednodušší prvek soft-procesoru a slouží k uchování aktuální hodnoty. V dané implementaci soft-procesor obsahuje 4 registry. Soft-procesor také obsahuje v/v porty, které jsou řešeny také pomocí registrů.

Multiplexer je funkční člen pracující na principu přepínače. V závislosti na vstupních řídících datech označených jako *source* vybere data z požadovaného vstupu a jeho hodnotu zobrazí na výstupu *out*. Tento multiplexer je využíván pro čtení hodnot z registrů, např. pro ALU.



Obrázek 55 - Zjednodušené schéma multiplexeru.

Demultiplexer má obrácenou funkci než multiplexer. Z jednoho datového vstupu je pomocí řídících dat označených jako *destination* zapsána hodnota na požadovaný výstup, např. zápis hodnoty do příslušného registru.



Obrázek 56 - Zjednodušené schéma demultiplexeru.

Paměť programu má v základu velikost 256 bajtů a skládá se ze dvou souborů. Základní soubor s označením rom_memory.v obsahuje modul, který se chová jako standardní ROM paměť. Pro svou funkci importuje textový soubor memory.rom obsahující posloupnost instrukcí v binárním tvaru, které generuje kompilátor CO8 ze zdrojového kódu (soubor s příponou ASM).

7.4 POPIS INSTRUKČNÍHO SOUBORU

Instrukční soubor je sada všech instrukcí, které procesor umí vykonat. Instrukce se obvykle rozdělují podle druhu operace, kterou vykonávají, např. aritmetické, logické, větvení programu, apod.

Navržený soft-procesor obsahuje 30 instrukcí, které mají konstantní délku (16 bitů) podobně jako u RISC, což umožňuje snadné a rychlé dekódování. Seznam instrukcí je rozdělen do 6 bloků, viz obrázek č.57. Pět nejvyšších bitů kódu určuje typ instrukce, zbývající nižší bity jsou parametry instrukce.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ostatní operace																
RESET	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Operace přesunu																
MOV Rx #K	1	0	1	0	1	k	k	k	k	k	k	k	k	r	r	r
MOV Rx Rx	0	0	1	0	1	0	0	0	0	0	r	r	r	r	r	r
MOV Rx Inx	0	0	1	0	1	0	0	0	0	0	1	0	x	r	r	r
MOV OUTx Rx	0	0	1	0	1	0	0	0	0	0	r	r	r	1	0	x
Aritmetické operace																
ADD Rx #K	1	0	0	0	1	k	k	k	k	k	k	k	k	r	r	r
SUB Rx #K	1	0	0	1	1	k	k	k	k	k	k	k	k	r	r	r
ADD Rx Rx Rx	0	0	0	0	1	0	0	0	r	r	r	r	r	r	r	r
SUB Rx Rx Rx	0	0	0	1	1	0	0	0	r	r	r	r	r	r	r	r
ADDC Rx Rx Rx	0	0	0	1	0	0	0	0	r	r	r	r	r	r	r	r
SUBC Rx Rx Rx	0	0	1	0	0	0	0	0	r	r	r	r	r	r	r	r
INC Rx	1	0	0	0	1	0	0	0	0	0	0	0	1	r	r	r
DEC Rx	1	0	0	1	1	0	0	0	0	0	0	0	1	r	r	r
Operace rotace a posuny																
RR Rx Rx	0	1	1	0	0	0	0	0	0	0	r	r	r	r	r	r
RL Rx Rx	0	1	1	0	1	0	0	0	0	0	r	r	r	r	r	r
RRC Rx Rx	0	1	1	1	0	0	0	0	0	0	r	r	r	r	r	r
RLC Rx Rx	0	1	1	1	1	0	0	0	0	0	r	r	r	r	r	r
SHL Rx Rx	0	1	0	1	0	0	0	0	0	0	r	r	r	r	r	r
SHR Rx Rx	0	1	0	1	1	0	0	0	0	0	r	r	r	r	r	r
Podmíněné a nepodmíněné skoky, Volání a návraty z podprogramů																
JMP rel	1	1	1	1	0	k	k	k	k	k	k	k	k	0	0	0
JZ rel	1	1	1	0	1	k	k	k	k	k	k	k	k	0	0	0
CALL	není implementována															
RET	není implementována															
Logické operace																
NEG Rsx	0	0	1	1	0	0	0	0	0	0	r	r	r	r	r	r
CLR Rsx	0	0	0	0	0	0	0	0	0	0	r	r	r	r	r	r
AND Rx #K	1	0	1	1	1	k	k	k	k	k	k	k	k	r	r	r
OR Rx #K	1	1	0	0	0	k	k	k	k	k	k	k	k	r	r	r
XOR Rx #K	1	1	0	0	1	k	k	k	k	k	k	k	k	r	r	r
AND Rx Rx Rx	0	0	1	1	1	0	0	0	r	r	r	r	r	r	r	r
OR Rx Rx Rx	0	1	0	0	0	0	0	0	r	r	r	r	r	r	r	r
XOR Rx Rx Rx	0	1	0	0	1	0	0	0	r	r	r	r	r	r	r	r

Obrázek 57 - Instrukční soubor soft-procesoru.

7.5 KOMPILÁTOR SOFT-PROCESORU

Kompilátor CO8 je napsán v jazyce Python a provádí jednoduché parsování vstupního souboru (assembler), z tohoto důvodu nejsou povoleny komentáře. Příkaz a jednotlivé operandy jsou odděleny mezerami, pro parsování se využívá metoda *Word(alphanums)*, která vrací jak písmena tak i číslice. Pokud požadujeme další znaky, např. u návěstí dvojtečku, přidáme je do parsovací metody pomocí konstrukce *+":"*. V případě chyby při parsování zdrojových dat překladač po zachycení výjimky vypíše chybové hlášení a ukončí se. Výstupní soubor *memory.rom* generovaný z překladače se zkopíruje do projektu a celý projekt se zkompiluje ve vývojovém prostředí Quartus nebo pomocí příkazové řádky. Následuje buď naprogramování FPGA nebo spuštění simulace. Výsledky simulace se graficky zobrazují pomocí aplikace GTKWave. Syntaxe příkazu se skládá z názvu kompilátoru *CO8.py*, parametr příkazu obsahuje název zdrojového kódu bez přípony *asm,* viz obrázek č. 58. Kompilátor využívá modul *pyparsing*, který načtený řádek ze souboru parsuje na příkaz a na operandy. V závislosti na vyčtených hodnotách se ještě rozlišuje, zda se jedná o návěští nebo číselnou konstantu. Instrukční soubor včetně operandů je uložen v samostatném souboru *instrukce.py* a je do hlavního programu importován direktivou *import*. Instrukce jsou v programu rozděleny podle počtu operandů a jsou dekódovány na 16bitový strojový kód. Tento kód je pak přímo zpracováván soft procesorem.

```
./CO8.py test_alu

START:
Navesti: START na radku c.: 0
MOV R1 #10
0xA881   1010100010000001
MOV R2 #01
0xA80A   1010100000001010
CLR R3
0x001B   0000000000011011
ADD R1 R1 R3
0x0859   0000100001011001
```

Obrázek 58 - Ukázka výpisu kompilace.

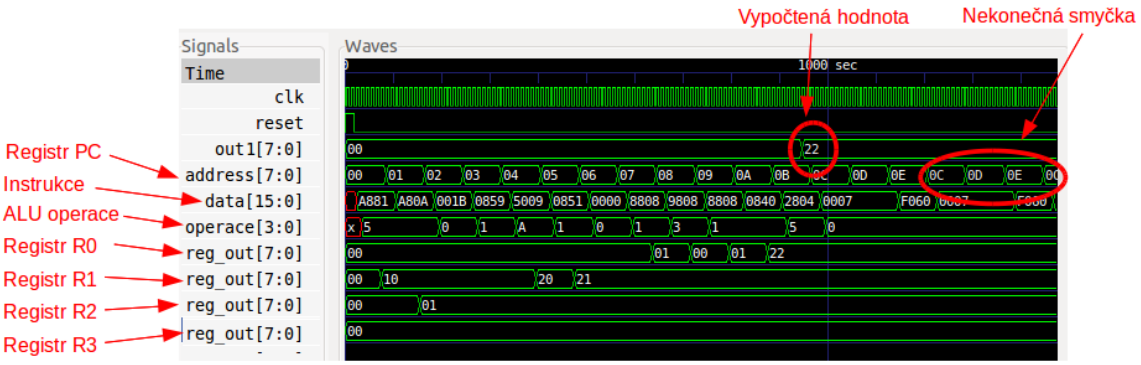
7.6 ŘEŠENÉ ÚLOHY SE SOFT-PROCESOREM

Ukázkové úlohy jsou zaměřeny na ověření funkčnosti vlastního návrhu soft-procesoru včetně ověření funkčnosti překladače. První úloha testuje funkčnost jednotky ALU. Program provede počáteční inicializaci registrů a následují aritmetické operace.

Překladač není „case sensitive“, proto je ve zdrojovém kódu tato vlastnost otestována. Výpis výstupu z překladače obsahuje vždy instrukci, její kód jak v hexadecimálním tak i binárním tvaru. U návěstí je vždy vypsána informace o čísle řádku. Další úloha testuje soft procesor v reálné úloze jako je silniční semafor. Obě úlohy jsou prakticky odzkoušeny na modulu DE0-Nano.

7.6.1 TESTOVÁNÍ ALU

Testování ALU se provádí krátkým programem, který otestuje nejen zmíněnou ALU ale i funkčnost celého soft-procesoru. Výsledná hodnota výpočtu je zobrazena na výstupním portu OUT1. Testovací program je uložen v souboru *test_alu.asm*. Ten je pak příkazem *./CO8.py test_alu* zkompilován a vygeneruje výstupní soubor *memory.rom*, obsahující strojové instrukce. Pro účely kontroly a případného hledání chyb je dobré si uchovat i výpis průběhu kompilace, což provedeme příkazem *./CO8.py test_alu >preklad.txt*. Soubor se strojovým kódem pak překopírujeme do adresáře SoftCPU, který obsahuje zdrojové verilog moduly procesoru. Po spuštění skriptu *runall.sh* se provede zkompilování verilog souborů, spuštění simulace a následné grafické zobrazení průběhu simulace programem GTKWawe, viz obrázek č. 59.

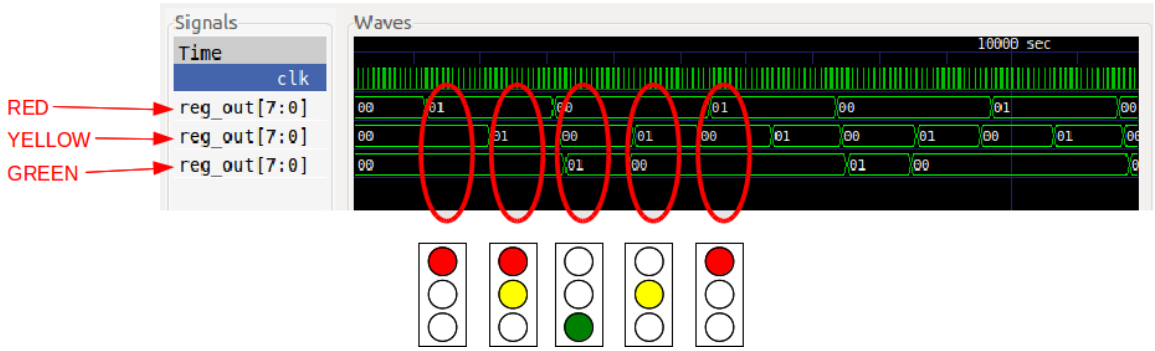


Obrázek 59 - Zobrazení průběhu simulace.

7.6.2 SILNIČNÍ SEMAFOR

Testovací úloha řeší vytvoření běžného silničního semaforu. V této části se otestuje funkčnost podmíněných skoků a časových smyček. Řešení úlohy je rozděleno na dvě části. První řešení je obdobou předchozí úlohy, tj. program je otestován pomocí simulace, viz obrázek č. 60. Druhé řešení je vytvoření projektu v prostředí Quartus a program je otestován přímo na kitu DE0-Nano. Z důvodu vysoké frekvence oscilátoru (50 MHz)

a způsobu připojení LED diod je program upraven a uložen do souboru *test_sem2.asm*. Dobu trvání jednotlivých stavů lze ovlivnit hodnotou v registru R3, která určuje počet opakování čekací smyčky.



Obrázek 60 - Zobrazení průběhu simulace.

8 NIOS

Procesor Nios II je navržen v jazyce HDL a patří mezi nejvíce univerzální soft-procesory na bázi FPGA. Je podporován všemi obvody SoC a FPGA od firmy Altera. V současné době se skládá ze tří procesorových jader, které mají společnou architekturu instrukční sady, každé je však optimalizováno pro konkrétní poměr cena/výkon.

Pomocí vývojového nástroje Qsys (dříve SOPC Builder) navrhujeme vlastní soft-procesor a k němu si vybíráme jen ty periferie, které jsou v projektu potřebné. Procesor s periférierni komunikuje pomocí sběrnice Altera Avalon, která umožňuje k procesoru připojit jakoukoliv námi definovanou entitu.

Typy jader procesoru:

- Nios II/f - jádro navržené pro vysoký výkon s možností široké konfigurace a optimalizace výkonu ,
- Nios II/e - jádro je navrženo tak, aby bylo co nejmenší na úkor některých omezení, jako je omezená sada instrukcí, omezená možnost nastavení jádra,
- Nios II/s – standardní jádro malé velikosti se zachováním výkonu vhodné pro většinu běžných aplikací.

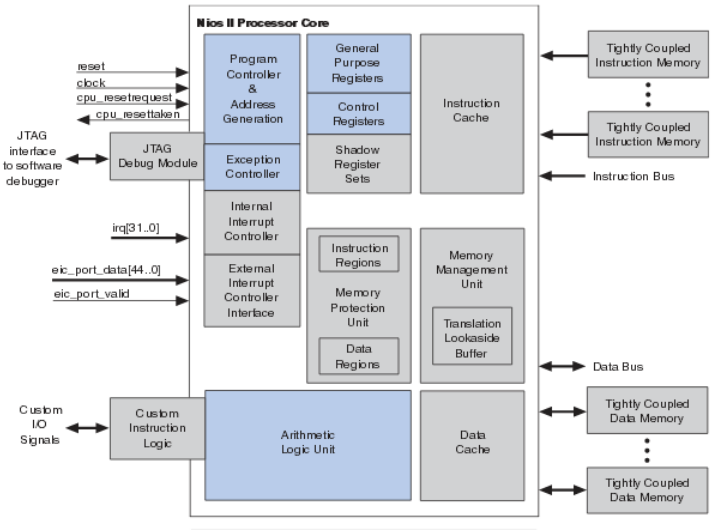
Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

Obrázek 61 - Odezva na přerušení (údaje v počtu hodinových cyklů). Zdroj [6].

Parametry jádra Nios II/e jsou:

- 32 –bitová RISC architektura,
- instrukční soubor obsahuje 256 instrukcí,
- schopnost adresace až 2 GB adresního prostoru,
- JTAG debug rozhraní,
- používá 600-700 logických prvků,
- absence MMU (memory management unit),
- dosahuje výkonu 30 DMIPS při 200 MHz.

Pro účely diplomové práce bylo zvoleno jádro Nios II/e, neboť ho lze používat bez potřebné licence, tj. je k dispozici zdarma.



Obrázek 62 - Nios II Procesor Core. Zdroj [6].

8.1 PIO KOMPONENTA

V této kapitole podrobně popíšeme tvorbu soft-procesoru NIOS a komponenty PIO. Tato komponenta bude konfigurována jako digitální výstupy, na které budou připojeny LED diody. V programovém prostředí Eclipse následně napíšeme testovací aplikaci pro otestování celého projektu.

Postup vytvoření projektu:

1. Spustíme Quartus II a založíme nový projekt.
2. Zadáme název projektu a umístění projektu.
3. Zvolíme řadu a typ FPGA: Cyclone IV E,EP4CE22F17C6.
4. Pro simulaci zvolíme jazyk Verilog a projekt vytvoříme.
5. Vytvoříme nový soubor (Schematic File), ve kterém v závěru propojíme vygenerovaný soft-procesor NIOS s piny FPGA.

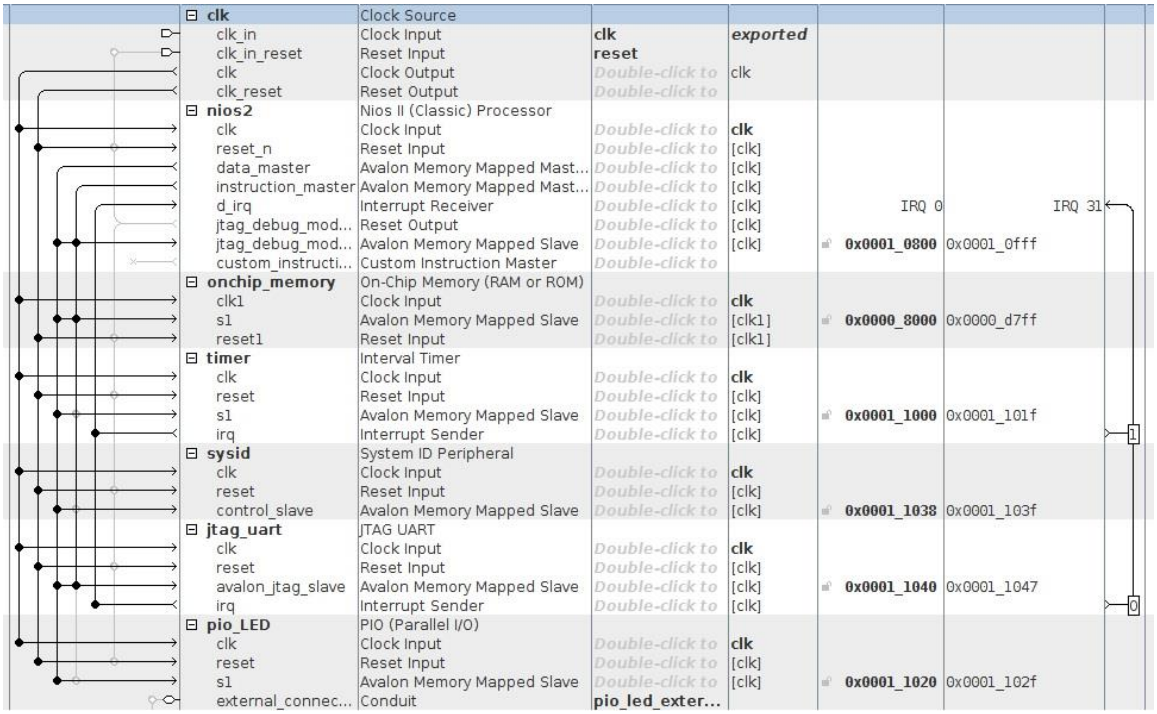
Pokračujeme v tvorbě soft-procesoru NIOS. Zde využíváme program QSYS, který umožňuje vytvářet libovolnou konfiguraci soft-procesoru včetně jeho periférií. Toto je nejdůležitější část návrhu.

6. Spustíme program QSYS z nabídky tools pro generování soft-procesoru NIOS.
7. Po spuštění máme v projektu automaticky přidanou komponentu CLK_0, kterou přejmenujeme na CLK.
8. Ostatní komponenty přidáme z nabídky knihovna (Library).

9. Přidáme komponentu NIOS II (volba Embedded Processors), zvolíme typ NIOS II/e, přejmenujeme ji na nios2.
10. Přidáme paměť pro uložení programu (volba Memories and Memory Controllers-On-Chip), volba a přejmenujeme na onchip_memory. Nastavíme ji na typ RAM o velikosti 20 kB (hodnota 20480)
11. Přidáme systémový časovač (volba Peripherals-Microcontrolle Peripherals) a přejmenujeme ji na timer. Necháme nastavení periody na 1 ms.
12. Dále přidáme systémové ID pomocí komponenty System ID Peripheral (volba Peripheral-Debug and Performace) a přejmenujeme ji na sysid. Nastavíme ji na hodnotu např. 0x22.
13. Následuje přidání komponenty JTAG-UART potřebné pro ladění programu a případný výpis zpráv (volba Interface Protokols – Serial) a přejmenujeme ji na jtag_uart.
14. Následuje přidání periférií soft-procesoru, v našem případě pouze PIO komponenta (volba Peripherals-Microcontrolle Peripherals), přejmenujeme ji na pio_LED. Ponecháme defaultní nastavení (8-bit output).

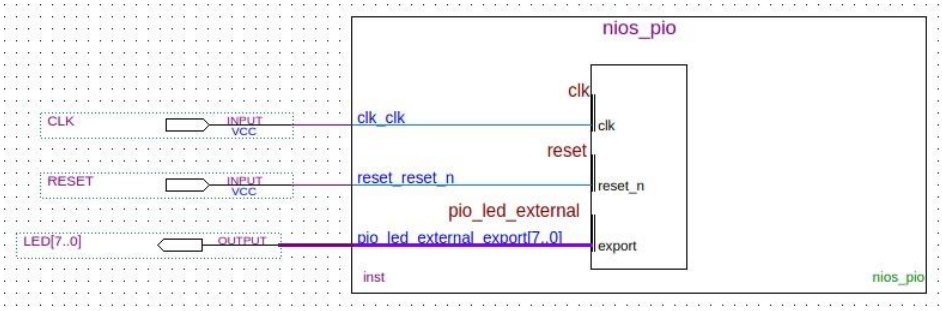
Nyní máme hotovou kostru soft-procesoru a musíme vzájemně propojit jednotlivé komponenty a nastavit adresní prostor těchto komponent.

15. U komponenty nios přesměrujeme vektory reset a exception do paměti onchip_memory.
16. Komponenty připojíme k interní sběrnici dle obrázku č. 63.
17. Necháme automaticky nastavit adresní prostory komponent (volba System- Assign Base Adresses).
18. Doplníme přerušení komponent – timer, jtag-uart.
19. Automaticky necháme nastavit čísla přerušení (volba System- Assign Interrupt Numbers).
20. Nastavíme external connection u komponenty pio_LED.
21. Vygenerujeme komponentu soft-processor (volba Generate).



Obrázek 63 - Ukázka programu QSYS.

Následujícími kroky v prostředí Quartus do schématu vložíme nově vytvořenou komponentu soft-procesoru a propojíme ji s fyzickými piny FPGA. Pro náš účel stačí hodinový signál CLK, tlačítko RESET a 8 x LED. Vzniklou konfiguraci zapíšeme do FPGA pomocí programátoru. Pak přejdeme k programové části a vytvoříme jednoduché počítadlo zobrazující hodnotu pomocí LED v binárním tvaru.



Obrázek 64 - Ukázka blokového schématu.

22. V prostředí Quarus provedeme propojení s fyzickými piny (Assigments Pin Planner).
23. Do projektu vložíme soubor (nios_pio/synthesis/nios_pio.qip).
24. Spustíme kompilaci a naprogramujeme FPGA.
25. Spustíme program Eclipse pro vytvoření programu.
26. Vytvoříme nový projekt pomocí BSP (Board Support Package).
27. Vybereme SOPC soubor a zadáme název projektu, volba prázdný projekt.
28. Vytvoříme soubor main.c a zkopírujeme kód z obrázku č. 65.

29. Nastavíme parametry projektu BSD (Nios II BSD Properties -Reduced device drivers, Small C library).
30. A nakonec program spustíme (volba Nios II Hardware).

```
#include <stdio.h>
#include "system.h"
#include "unistd.h"
#include "altera_avalon_pio_regs.h"

int main()
{
printf("Pripojen k DE0 nano\n");
printf("Test LED ..\n");

int  count = 0;

while(count<256)
{
 IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, count );
 usleep(200000); // sleep 0,2s
 count++;
}

printf("Test ukoncen. \n");
return 0;
}
```

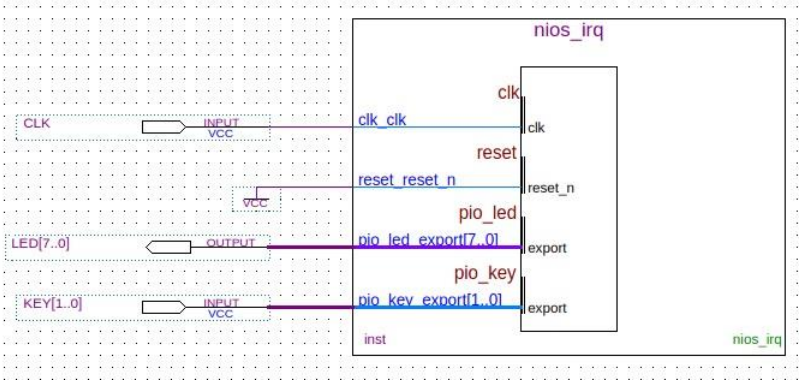
Obrázek 65 - Ukázka programu komunikace s PIO rozhraním.

8.2 IRQ

Ukázka využití přerušení pro detekci stisku tlačítek vychází z předchozího projektu PIO, pouze je přidána komponenta PIO (přejmenovaná na PIO_KEY) nastavená jako 2bitový vstup s detekcí sestupné hrany pro přerušení. Datový vstup komponenty je pojmenován stejně jako komponenta (pio_key). V blokovém schématu je pak propojen se vstupem KEY[1..0]. Značení v hranatých závorkách označuje sběrnici o dvoubitové šířce.

Obrázek 66 - Ukázka programu QSYS.

Vzhledem k opakování stejných komponent v návrhu QSYS (identický návrh) bude v následujících ukázkách zobrazena jen ta část komponent, kde dochází ke změnám, tj. od komponenty jtag_uart, viz obrázek č. 66.



Obrázek 67 - Ukázka blokového schématu.

Předchozí program z obrázku č. 65 je modifikován na čítač impulsů, je přidána registrace přerušovací rutiny a obslužná rutina pro přerušení, viz obr. 68. Po stisku tlačítka KEY0 je inkrementována hodnota vnitřního čítače. Jeho hodnota je v binárním tvaru zobrazena pomocí LED diod. Stiskem tlačítka KEY1 je vnitřní čítač vynulován.

```
#include <stdio.h>
#include "system.h"
#include "unistd.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include <sys/alt_irq.h>

int pocitadlo;

static void key_handle()
{
    int key=IORD _PIO_EDGE_CAP (PIO_KEY_BASE);
    if(key==2) { printf("-> interrupt KEY0 - increment \n");
                 pocitadlo++;}
    if(key==1) { printf("-> interrupt KEY1 - reset\n");
                 pocitadlo=0;}

    // reset edge capture register
    IOWR_PIO_EDGE_CAP(PIO_KEY_BASE, 0);

    //zobraz hodnotu
    IOWR_PIO_DATA(PIO_LED_BASE, pocitadlo);
}

int main()
{
      pocitadlo=0;
      printf("Pripojen k DE0 nano\n");
      printf("Test IRQ ..\n");

      printf("* cti tlacitka za 2 sec\n");
      usleep(2000000);
      int cti= IORD_PIO_DATA(PIO_KEY_BASE);
      printf(" key value: %d \n",cti);
      printf(" OK \n");

      printf("\n* test ISR \n");
      // povol preruseni pro obe tlacitka//
      IOWR_ PIO_IRQ_MASK(PIO_KEY_BASE, 0x03);
      // reset edge capture register
      IOWR_ PIO_EDGE_CAP(PIO_KEY_BASE, 0x0);
      // registrace ISR
      alt_irq_register( PIO_KEY_IRQ, NULL, key_handle );

      printf("Testuj IRQ: \n");
      return 0;
}
```

Obrázek 68 - Ukázka programu použití IRQ.

8.3 UART KOMPONENTA

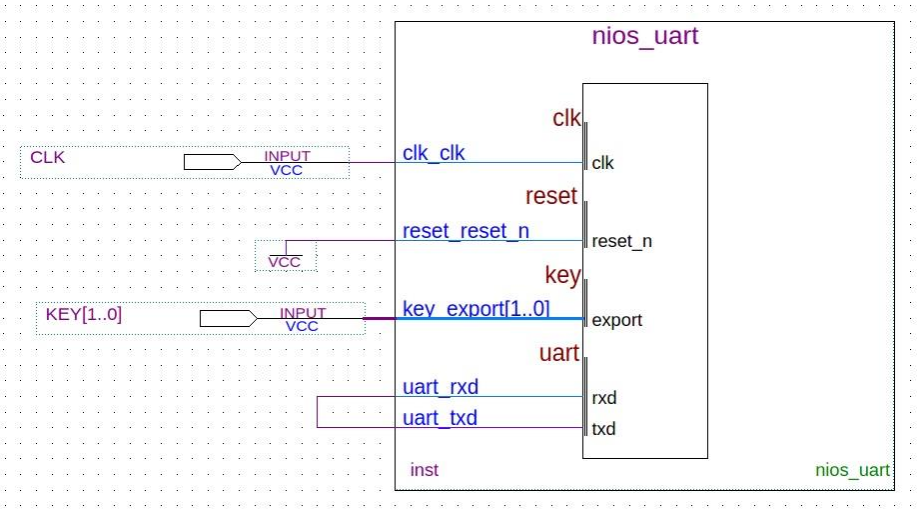
Projekt slouží k otestování sériového rozhraní UART za použití smyčky (loop back). Zde je opět rozšířena předchozí úloha o komponentu UART, která je nakonfigurovaná na přenosovou rychlost 9600 b/s. Tato komponenta má povolené přerušení pouze pro příjem dat.

Obrázek 69 - Ukázka programu QSYS.

Vlastní smyčka je realizována již ve schématu, jako propojení výstupu TxD se vstupem RxD, proto při přiřazování vstupů/výstupů k fyzickým pinům FPGA se tyto názvy nezobrazují. V případě potřeby sériovou komunikaci zobrazit, pomocí osciloskopu nebo logického analyzátoru, můžeme tento signál vyvést na fyzické piny (output).

```
#include <string.h>
#include <unistd.h>
#include "system.h"
#include "alt_types.h"
#include "altera_avalon_uart_regs.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"

static void key_handle()
{
    int key=IORD_ PIO_EDGE_CAP(PIO_KEY_BASE);

    //posli znak
    if(key==2) { printf("-> interrupt KEY0 - send A \n");
    if(key==1) { printf("-> interrupt KEY1 - send B \n");

    // reset edge capture register
    IOWR_ PIO_EDGE_CAP(PIO_KEY_BASE, 0);
}

static void  uart_handle()
{
    unsigned short int data,status;

    printf("-> Interrupt UART ..\n");
    status = IORD_UART_STATUS(UART_BASE);
    printf("-> UART status: %d\n",status);

    data =IORD_UART_RXDATA(UART_BASE)& 0x0ff;
    printf("-> RX data: %d\n",data);
}

int main(void)
  {
    printf("Pripojen k DE0 nano\n");
    printf("Test UART ..\n");

    // povol preruseni pro obe tlacitka//
    IOWR_ PIO_IRQ_MASK(PIO_KEY_BASE, 0x03);
    // reset edge capture register
    IOWR_PIO_EDGE_CAP(PIO_KEY_BASE, 0x0);
    // registrace ISR
    alt_irq_register( PIO_KEY_IRQ, NULL, key_handle );

    printf("\n* test ISR \n");
    int divisor = (int)(50000000/9600+0.5);   //9600 b/s
    // nastaveni prenosove rychlosti
    IOWR_ UART_DIVISOR(UART_BASE, divisor);
    // nastaveni preruseni pro prijmu
    alt_u32 control = UART_CONTROL_RRDY_MSK;
    IOWR_UART_CONTROL(UART_BASE, control);
    // registrace ISR
    alt_irq_register(UART_IRQ, NULL, uart_handle);

    printf("Testuj UART: \n");
    return 0;
  }
```

Obrázek 70 - Ukázka programu UART.

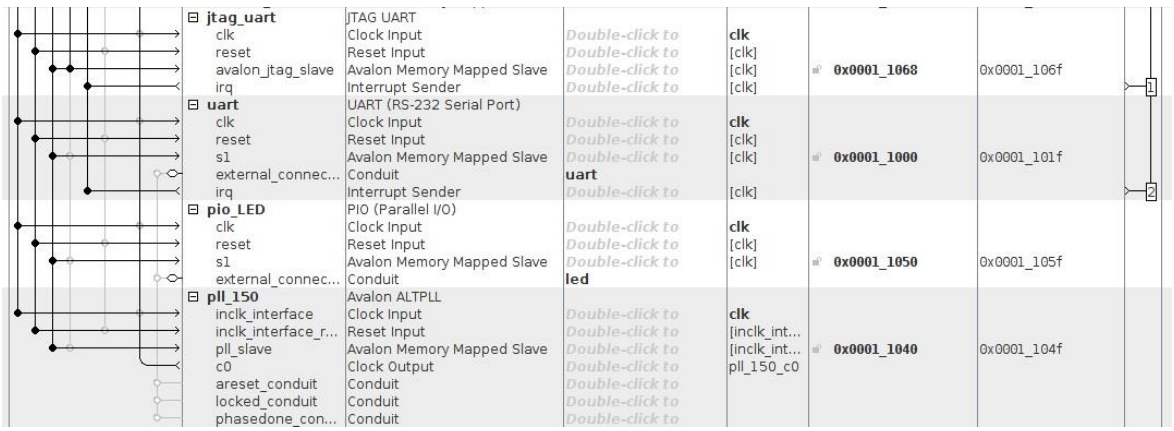


Obrázek 71 - Ukázka blokového schématu.

Aplikace čeká na stisk jednoho ze dvou tlačítek, která jsou pak následně obsloužena pomocí přerušení. V přerušovací rutině je otestováno, které tlačítko vyvolalo přerušení a podle výsledku je vyslán patřičný znak. V okamžiku příjmu znaku je vyvoláno přerušení komponenty UART a přijatá hodnota je zobrazena v konzoli, viz ukázka programu.

8.4 UART-USB

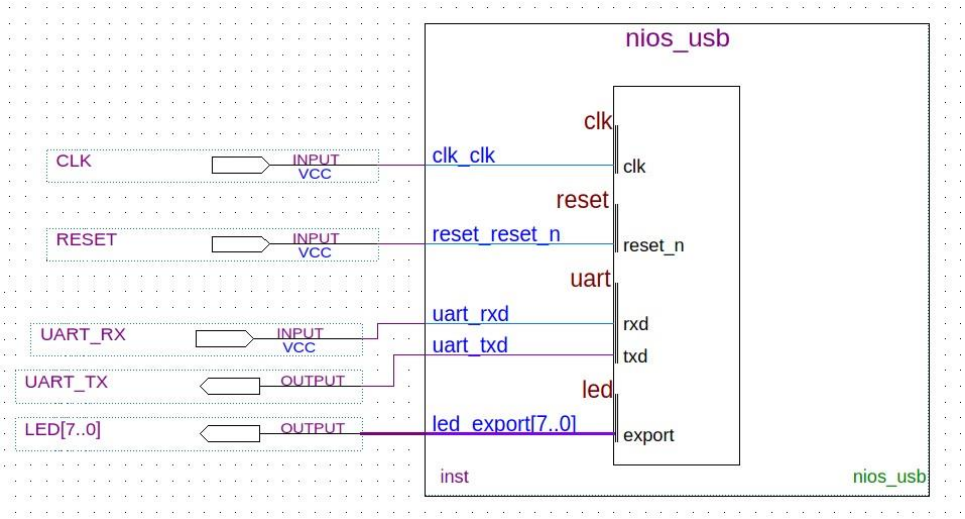
Tento projekt umožňuje komunikovat s PC pomocí rozhraní USB. Pro komunikaci je zde použita komponenta UART, neboť komponenta USB není obsažena v základní knihovně komponent. Možným řešením by byla možnost importovat externí USB komponentu, ale z důvodu licenčního omezení nebyla tato varianta použita. Pro převod mezi oběma rozhraními byl použit modul MM232R s obvodem FT232RQ.



Obrázek 72 - Ukázka programu QSYS.

V návrhu přibyla komponenta pll_150, ve které je implementována smyčka fázového závěsu (Phase Locked Loop). Tento fázový závěs nám zvyšuje trojnásobně

vstupní frekvenci (z 50 MHz na 150 MHz). Tato vyšší frekvence je pak využita jako taktovací frekvence soft-procesoru, což umožňuje rychlejší zpracování přijatých znaků.



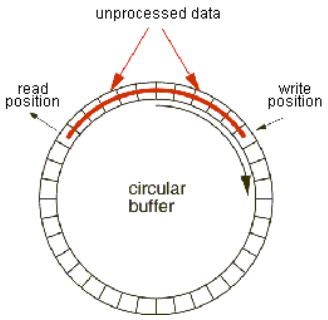
Obrázek 73 - Ukázka blokového schématu.

Program čeká na příchozí znak, po přijetí je volána rutina přerušení a znak je uložen do bufferu. Po přijetí určitého počtu znaků (uart_count_byte) je nastaven příznak uart_flag a zavolána funkce zpracuj_buff().

```
void zpracuj_buff()
{ //zpracuje "opraveny buffer"
  if(buff[0]!=0x05) return;
  printf("...Write Single Coil\n");

  int addr=buff[1]*256+buff[2];
  int value=buff[3]*256+buff[4];
  //zapis na PIO_LED
  alt_u8 pio=IORD_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE);
  alt_u16 bit=0x0001;
  bit=bit<< (addr & 0x0007);

  if(value==0x0000) pio=pio&(~bit);   //OFF
   else   pio=pio|bit;   //ON
  IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, pio);
}
```

Obrázek 74 - Ukázka části programu.

Zde se přijatý paket zpracuje a odešle se návratový paket. V příkladu je implementována Modbus funkce Write Single Coil a jako výstupy jsou použity LED diody na modulu DE0-Nano. Všechny tyto ukázky užití komponent slouží jako příprava na finální produkt NanoMod. Obslužný program v PC je opět napsán ve skriptovacím jazyce Python (obrázek č. 76).



Obrázek 75 - Zobrazení funkce kruhového bufferu. Zdroj [9].

Přijímané znaky jsou zapisovány do kruhového bufferu, který slouží jako vyrovnávací paměť. Kruhový buffer (Circular buffer) je implementací fronty (FIFO) nad polem a skládá se z pole fixní délky a dvou ukazatelů. Asymptotická složitost výběru a čtení prvku na prvním indexu je O(1), složitost operace přidání prvku na konec fronty je O(1).

```
#!/usr/bin/env python

from pylibftdi import Device
import codecs
import binascii

print("Modbus: 0x05-Write Single Coil")
b_array =bytearray([ 0x05, 0x00, 0x00, 0x00, 0x00])

with Device(device_id='FTDHNZSD',mode='t') as dev:
   dev.baudrate = 2400
   while(1):

         num=int(raw_input('Zadej vystup(0-7,9-konec):'))
         if(num==9):
               break
         b_array[2]=num
         value=int(raw_input('Zadej hodnotu(0-OFF,1-ON):'))
         if(value==1):
               b_array[3]=0xFF
         else:
               b_array[3]=0x00
         ss=binascii.hexlify(b_array)
         send=codecs.decode(ss, 'hex_codec')
         print("Send data: "+ss)
         dev.flush()
         dev.write(send)
```

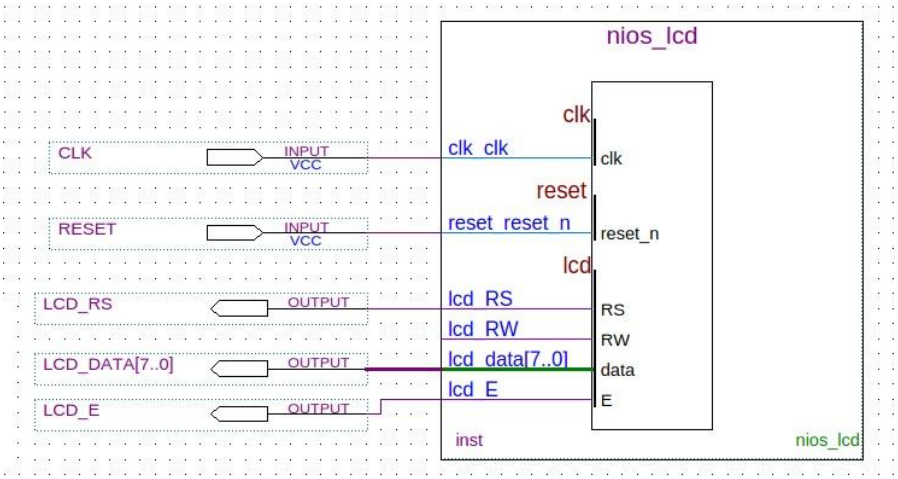
Obrázek 76 - Ukázka zdrojového kódu. Soubor: test_usb.py.

8.5 LCD KOMPONENTA

Projekt je zaměřen na zobrazování uživatelských dat na LCD displeji 4 x 20 znaků. Pomocí 3 vytvořených funkcí (LCD_Clear, LCD_SetPos, LCD_Print) lze zobrazit na displeji všechna potřebná data. Pro komunikaci se soft-procesorem NIOS je zvolena komponenta LCD Optrex 16207, která komunikuje i se všemi LCD zobrazovači s řadičem kompatibilním s HD44780.

Obrázek 77 - Ukázka programu QSYS.

Vzhledem k použití 5V logiky displeje není zapojen výstup RW, data do displeje jsou pouze zapisována. Tímto řešením odpadá nutnost realizace převodníku 5 V/3.3 V. Z pohledu programového je nutno tedy nahradit čtení příznaku BUSY čekací časovou smyčkou, dle doporučení výrobce displeje.



Obrázek 78 - Ukázka blokového schématu.

Pro komunikaci displeje s FPGA je použito 8 datových vodičů (8-mi bitový režim) a dva řídící signály. První slouží pro přepínání zápisu instrukcí/dat (RS), druhý slouží k potvrzení platných dat na sběrnici (E) sestupnou hranou.

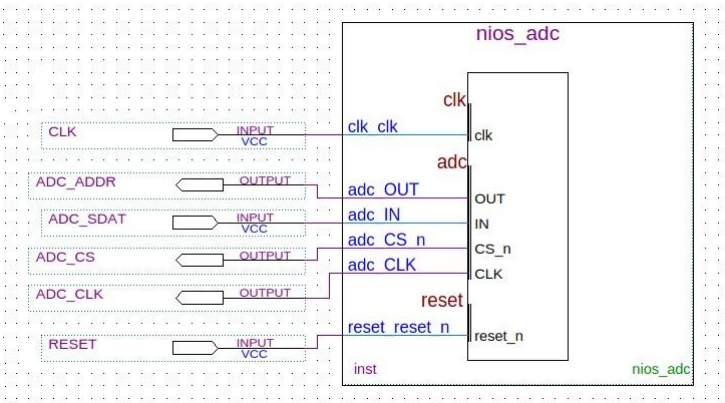
```
int main()
{
  char Text[20] = "Prvni radek";

  printf("Test LCD ..\n");

  LCD_Init();

  //radek, sloupec
  LCD_SetPos(0,0);    LCD_Print(Text);
  LCD_SetPos(1,0);   LCD_Print("Druhy radek");
  LCD_SetPos(2,0);   LCD_Print("Treti radek");
  LCD_SetPos(3,0);   LCD_Print("Ctvrty radek");

 //zobraz promnenou pom na 4 radek, na pozici 12 sloupce
 int pom=256138;
 char myBuff[20];
 sprintf(myBuff, "%i", pom);

 LCD_SetPos(3,11);  LCD_Print(myBuff);
…
}
```

Obrázek 79 - Ukázka zdrojového kódu.

8.6 ADC KOMPONENTA

Pro ovládání 12-bitového integrovaného AD převodníku je použita externí komponenta od firmy Terasic. Tato komponenta realizuje SPI komunikaci, která je přímo upravena pro převodník ADC128S022. Fázovy závěs pll_adc zde slouží pro vytvoření hodinového signálu o hodnotě 2 MHz pro AD převodník.

Obrázek 80 - Ukázka programu QSYS.



Obrázek 81 - Ukázka blokového schématu.

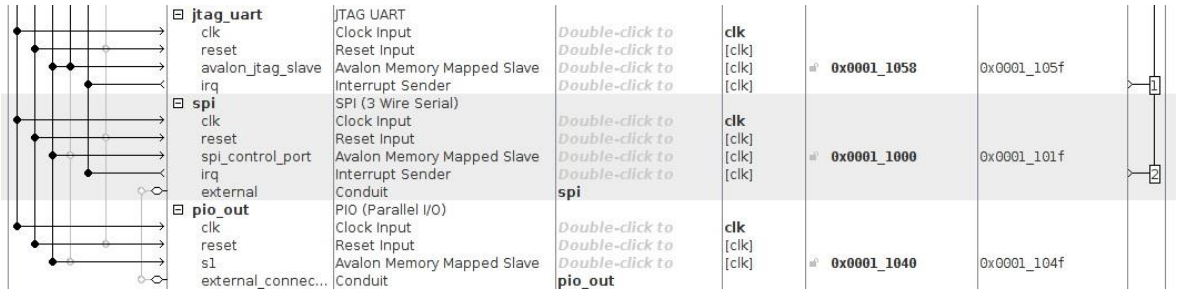
V aplikaci se zvolí požadovaný vstup AD převodníku, změří se z něho hodnota a následně přepočte na napětí. V případě 12-ti bitového převodníku se hodnota pohybuje v rozmezí 0 - 4095. Tato hodnota se pak vynásobí referenčním napětím převodníku (3.3 V).

```
int main(void)
{
  int ch = 1; //vstup

  while(1)
   {
    alt_u16  data16 = ADC_Read(ch); // 12-bits resolution
    float val=((float)data16) * 3.3 / 4.0950;
    printf("Vstup ADC: %d = %i mV\r\n", ch,(int)val);
   …
  }
  return 0;
}
```

Obrázek 82 - Ukázka části zdrojového kódu.

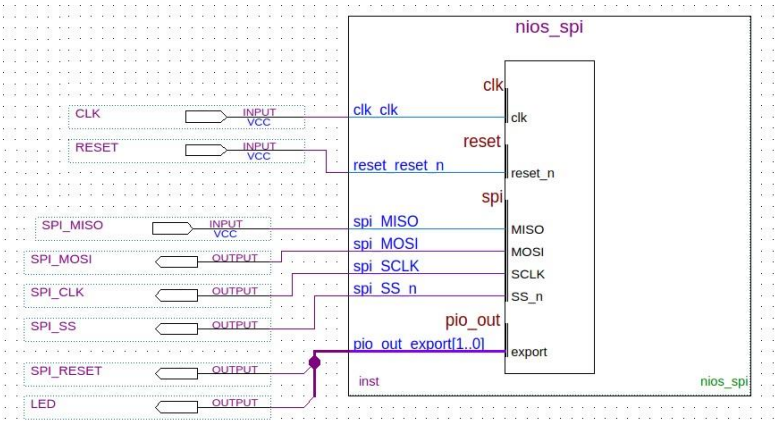
8.7 SPI KOMPONENTA

Pro otestování SPI rozhraní využijeme řadič ethernetu W5200, u kterého provedeme základní konfiguraci, nastavení MAC adresy a IP adresy. Správnou konfiguraci pak otestujeme pomocí příkazu ping mezi modulem a počítačem.



Obrázek 83 - Ukázka programu QSYS.

Modul WIZ820IO je osazen zmíněným ethernetovým řadičem W5200 obsahujícím integrovaný MAC i PHY (10/100 Mbps). Se soft-procesorem řadič komunikuje přes rozhraní SPI, které sice neumožňuje plně využít přenosovou rychlost ethernetu, ale pro potřeby protokolu Modbus TCP je dostačující. V případě nutnosti zvýšení datové propustnosti lze čip zaměnit za řadič W5300, který komunikuje po 16-ti bitové datové sběrnici.



Obrázek 84 - Ukázka blokového schématu.

```
int main(void)
{
  printf("Pripojen k DE0 nano\n");
  printf("Test SPI-W5200 ..\n");

  int i;
  //reset W5200
  IOWR_ PIO_DATA(PIO_OUT_BASE,0x00);
  usleep(50);
  IOWR_ PIO_DATA(PIO_OUT_BASE,0x01);
  for(i=0;i<20; i++) usleep(5000);

  //nastaveni MR
  alt_u8 mr[]={RST};
  write_SPI(MR,1,mr);
  usleep(5000);
  read_SPI(MR,1);
  while(resv[0] !=0) read_SPI(MR,1);
  printf("RE::MR: %02x\n",resv[0]);
```

```
//nastaveni MAC
alt_u8 mac[]=MAC;
write_SPI(SHAR,6,mac);
read_SPI(SHAR,6);
printf("MAC address:
        %02X.%02X.%02X.%02X.%02X.%02X\n",
        mac[0],mac[1],mac[2],mac[3],mac[4],mac[5]);
printf("RE::MAC address:
        %02X.%02X.%02X.%02X.%02X.%02X\n",
        resv[0],resv[1],resv[2],resv[3],resv[4],resv[5]);
  …
 //nastaveni IP adresy
alt_u8 ip[]=IP;
write_SPI(SIPR,4,ip);
read_SPI(SIPR,4);
printf("IP address: %u.%u.%u.%u\n",ip[0],ip[1],ip[2],ip[3]);
printf("RE::IP address:
        %u.%u.%u.%u\n",resv[0],resv[1],resv[2],resv[3]);
  …
 }
```

Obrázek 85 - Ukázka zdrojového kódu.

8.8 SPI-ETHERNET

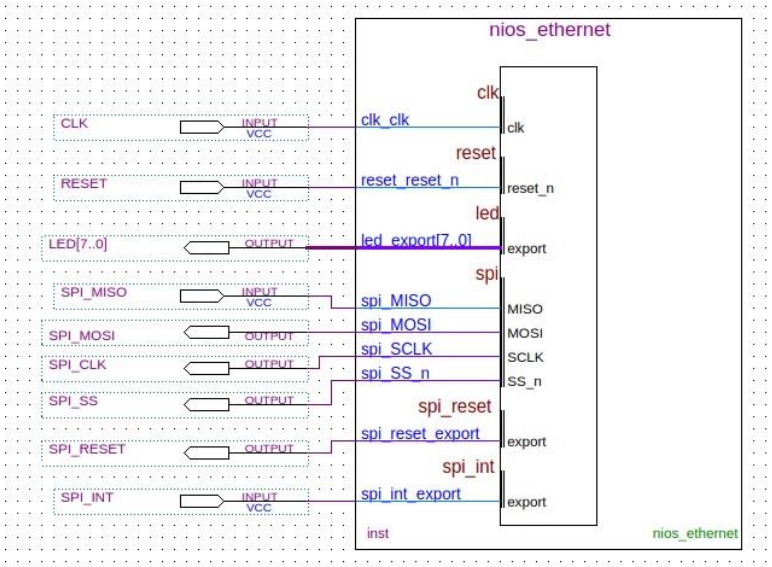
Projekt rozšiřuje předchozí ukázku SPI rozhraní o komunikaci pomocí UDP protokolu. Protokol UDP je nespojovaná služba, tj. nenavazuje spojení. Odesilatel pouze odešle UDP datagram příjemci a už se nestará, zda příjemce datagram obdržel nebo zda-li se datagram neztratil či ho obdržel vícekrát.

Z pohledu konstruovaného zařízení (slave) se bude provozovat UDP server, tj. zařízení bude očekávat povely od nadřízeného zařízení (master), v našem případě od PLC. Pro účely ladění programu jsou v Nios konzoli vypisovány příchozí pakety.

Obrázek 86 - Ukázka programu QSYS.

V ukázkovém příkladu je opět implementována Modbus funkce Write Single Coil. Zde oproti USB komunikaci se data neukládají do kruhového bufferu. Řadič již v sobě má

implementovaný 32 kB buffer, který se programově rozdělí mezi TX/RX sockety. V této ukázce je použita defaultní velikost 2 kB pro příjem a 2 kB pro vysílání.



Obrázek 87 - Ukázka blokového schématu.

Funkce Socket0_Recvfrom() vrací celý Modbus paket ADU, tj. jak hlavičku MBAP tak i PDU datovou část. Funkce ctiModbus() pak zpracovává pouze PDU část, MBAP hlavička se zde nezpracovává.

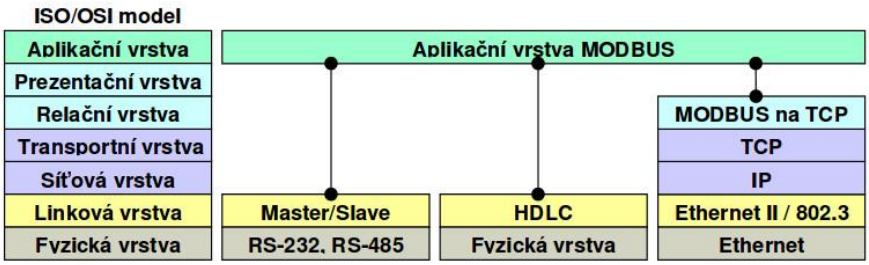
```
#include "system.h"
#include "alt_types.h"
#include "W5200.h"
#include "ethernet.h"
#include "modbus.h"

int main(void)
{
  printf("Pripojen k DE0 nano\n");
  printf("Test SPI-W5200 ..\n");

  Ethernet_Init();
  Socket0_Init();  //socket+bind
  alt_u8 * data;
  int res;
```

```
while(1)
  {
    //precti data
    data=Socket0_Recvfrom();

    //proved MODBUS prikaz
    res=ctiModbus(data);

    //posli odpoved
    if(res==0)
        res=Socket0_Sendto(data,5);
    else {
       data[0]=0x85;
       data[1]=0x01;
       res=Socket0_Sendto(data,2);
      }

    if(res!=0) printf("CHYBA TX timeout");
  }
return 0;
```

Obrázek 88 - Ukázka zdrojového kódu.

9 MODBUS

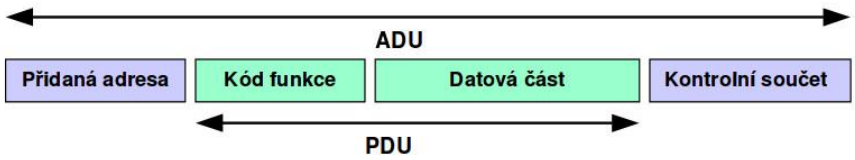
Modbus je průmyslový komunikační protokol na úrovni aplikační vrstvy modelu OSI. Umožňuje komunikaci typu klient-server mezi PLC a externími zařízeními (terminál, frekvenční měniče, ...). Protokol byl vytvořen v roce 1979 firmou Modicon a podporuje řadu komunikačních médií (RS-232, RS-485, Ethernet). Pro účely této práce je využita verze Modbus TCP, která využívá síť Ethernet. Podrobný popis protokolu je obsažen v [15].



Obrázek 89 - Příklad implementace protokolu. Zdroj [15].

9.1 POPIS PROTOKOLU

Protokol MODBUS definuje strukturu zpráv na úrovni protokolu (PDU - Protokol Data Unit) nezávisle na typu komunikační vrstvy. PDU je pak rozšířena o další části závislé na typu sítě a tvoří zprávu na aplikační úrovni (ADU – Application Data Unit). PDU se skládá ze dvou částí a to z kódu funkce a datové části. Kód funkce udává typ operace, její rozsah je 1-255, přičemž kódy 128-255 jsou vyhrazené pro chybové stavy. Obsah datové části je nepovinný a může obsahovat například adresu v paměti, hodnotu registru pro danou operaci.



Obrázek 90 - Základní tvar MODBUS RTU zprávy. Zdroj [15]

Komunikace probíhá ve třech krocích metodou požadavek-odpověd:

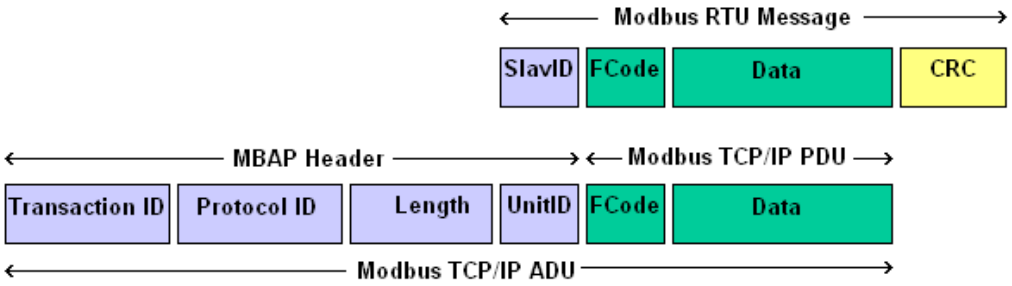
1) klient odešle požadavek skládající se z kódu funkce a dat požadavku,

2) server tento požadavek přijme a provede/neprovede požadovanou operaci,

3) server odpoví klientovi, odpověď se skládá z kódu funkce a dat odpovědi. V případě neúspěšného provedení operace je nastaven nejvyšší bit v kódu funkce (+0x80).

Protokol MODBUS využívá tzv. „Big-Endian“ prezentaci dat, což znamená, že nejvyšší byte je posílán jako první a nejnižší jako poslední.

ADU, jak již bylo zmíněno, se liší podle typu sítě. U verze MODBUS RTU je zpráva rozšířena o identifikační číslo a kontrolní součet. U verze MODBUS TCP je zpráva rozšířena o MBAP (Modbus Application Header) hlavičku obsahující např. ID transakce, délku zprávy, apod.



Obrázek 91 - Struktura protokolu Modbus. Zdroj [15].

9.2 DATOVÝ MODEL

Datový model MODBUS bývá nejčastěji realizován se čtyřmi oddělenými bloky, viz obrázek č. 92. Mapování tabulek do adresního prostoru je závislé na konkrétním zařízení. Každá z tabulek může mít vlastní adresní prostor nebo se mohou částečně či úplně překrývat.

Z důvodu zpětné kompatibility bývá adresní prostor rozdělen na bloky o velikosti 10000 položek, viz sloupec Adresa. Přístupná je každá položka jednotlivě nebo lze přistupovat ke skupině položek najednou. Velikost skupiny položek je omezena maximální velikostí datové části zprávy.

Blok	Typ	Přístup	Adresa	Popis
Diskrétní vstupy	1-bit	R	10000-19999	Data poskytuje I/O systém
Cívky	1-bit	R/W	0-9999	Data modifikovatelná programem
Vstupní registry	16-bit	R	30000-39999	Data poskytuje I/O systém
Uchovávací registry	16-bit	R/W	40000-49999	Data modifikovatelná programem

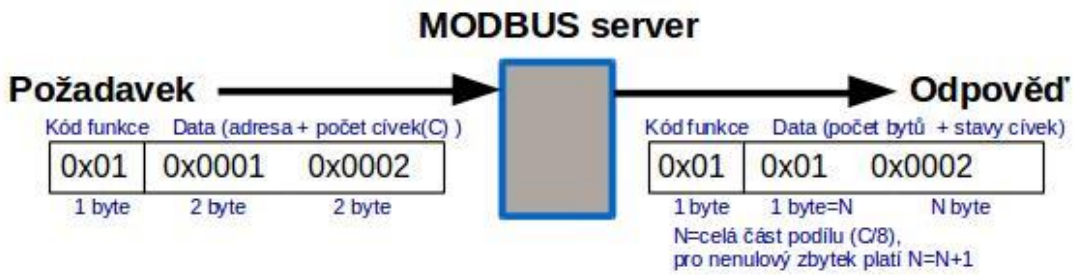
Obrázek 92 - Datový model Modbus.

9.3 POPIS KÓDŮ FUNKCÍ

V následující části budou popsány pouze kódy funkcí, které budou realizovány na modulu NanoMod. Pro otestování funkčnosti těchto kódů jsou vytvořeny skripty v jazyce Python nesoucí shodné jméno jako testované funkce, např. pro funkci čtení cívky bude skript mít název *ReadCoils.py*. O další funkce lze nanoMod jednoduše rozšířit modifikací metody *ctiModbus()*. Toto platí i pro rozšíření datového modelu – zvětšení počtu vstupů/výstupů, připojení dalších snímačů apod.

9.3.1 ČTI CÍVKY (READ COILS)

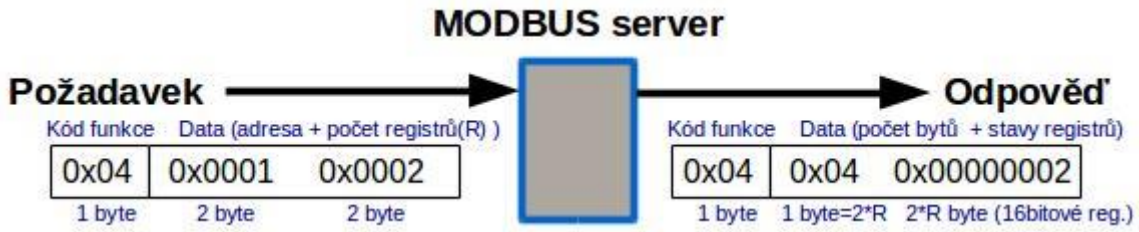
Tato funkce slouží ke čtení stavu 1 až 2000 cívek. V požadavku specifikujeme kód funkce (1 bajt – 0x01), adresu první cívky (2 bajty) a počet cívek (2 bajty). V odpovědi jsou stavy cívek přenášeny po oktetech (stav 8 cívek), nejnižší bit specifikuje stav první adresované cívky. Tato funkce je v modulu NanoMod použita pro čtení digitálních galvanicky oddělených vstupů.



Obrázek 93 - Ukázka protokolu Modbus.

9.3.2 ČTI VSTUPNÍ REGISTRY (READ INPUT REGISTERS)

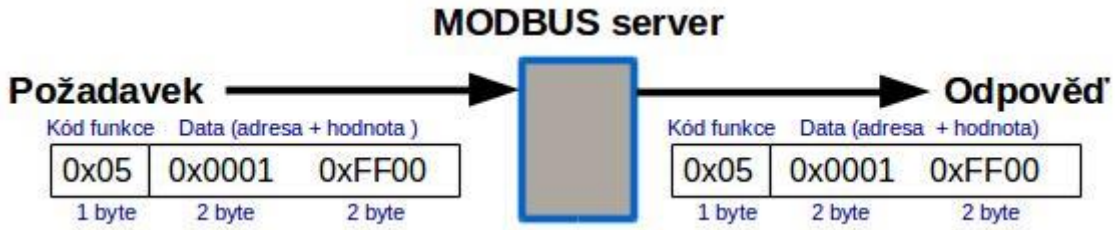
Tato funkce slouží ke čtení obsahu souvislého bloku až 125 vstupních registrů. V požadavku je specifikována adresa prvního registru a počet registrů. V odpovědi odpovídá každému registru dvojice bytů. Tato funkce je v modulu NanoMod použita pro čtení periférie ADC.



Obrázek 94 - Ukázka protokolu Modbus.

9.3.3 ZAPIŠ JEDNU CÍVKU (WRITE SINGLE COIL)

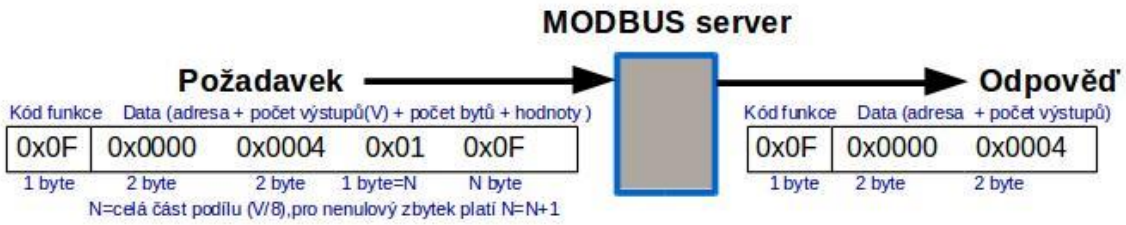
Tato funkce slouží k nastavení jednoho výstupu cívky do stavu ON nebo OFF. V požadavku je specifikována adresa výstupu, který se má nastavit, a hodnota, na kterou se má nastavit. Hodnota 0x0000 znamená OFF, hodnota 0xFF00 znamená ON.



Obrázek 95 - Ukázka protokolu Modbus.

9.3.4 ZAPIŠ VÍCE CÍVEK (WRITE MULTIPLE COILS)

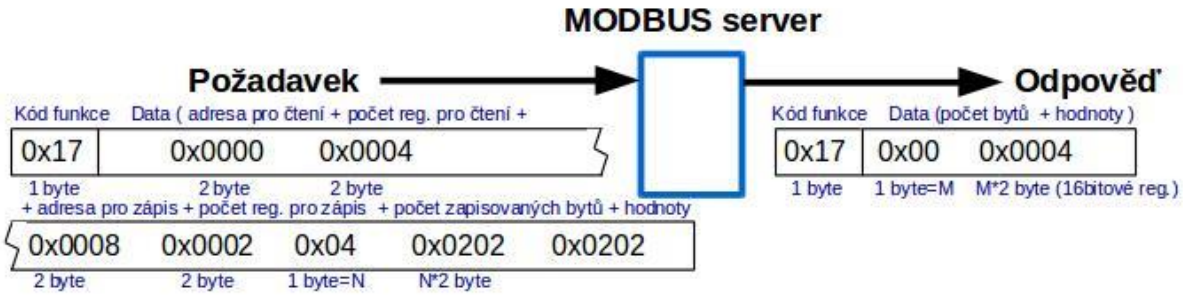
Tato funkce slouží k nastavení 1 až 1968 cívek. V požadavku specifikujeme kód funkce (1 bajt – 0x0F), adresu prvního výstupu (2 bajty), počet výstupů (2 bajty, max 0x7B0) a hodnoty, na které se mají výstupy nastavit. Tato funkce je v modulu NanoMod použita pro nastavení digitálních galvanicky oddělených výstupů.



Obrázek 96 - Ukázka protokolu Modbus.

9.3.5 ČTI/ZAPIŠ VÍCE REGISTRŮ (READ/WRITE MULTIPLE REGISTERS)

Tato funkce provádí kombinaci čtení a zápisů registrů v jedné transakci. Operace zápisu je provedena před operací čtení. V požadavku specifikujeme kód funkce (1 bajt – 0x17), počáteční adresu pro čtení (2 bajty), počet registrů pro čtení (2 bajty, max 0x76), počáteční adresu pro zápis (2 bajty), počet registrů pro zápis (2 bajty, max. 0x76) a hodnoty, které se mají zapsat.



Obrázek 97 - Ukázka protokolu Modbus.

10 NANOMOD

Předmětem diplomové práce je navrhnout a realizovat zařízení Modbus TCP na technologii FPGA. V předchozích kapitolách jsou popsány všechny funkční bloky, ze kterých se bude výsledné zařízení skládat. Název zařízení NanoMod byl zvolen záměrně, jako složení dvou prvků, a to vývojový kit DE0-Nano a protokol Modbus.

Zařízení je navrženo jako open source za použití volně dostupných návrhových a vývojových prostředků. Pro snadný vývoj byl použit modulární systém, tj. více desek PS vzájemně propojených pomocí propojovacích kablíků. Toto řešení umožňuje v budoucnu dále systém rozšiřovat o další komponenty.

Blok	**Typ**	**Přístup**	**Adresa**	**Popis**
Vstupy	1-bit	R	0x00 - 0x3F	použito16 optočlenů
Výstupy	1-bit	R/W	0x40 – 0x7F	použito 16 relé
ADC	16-bit	R	0x80 – 0xBF	použito 8 analogových vstupů

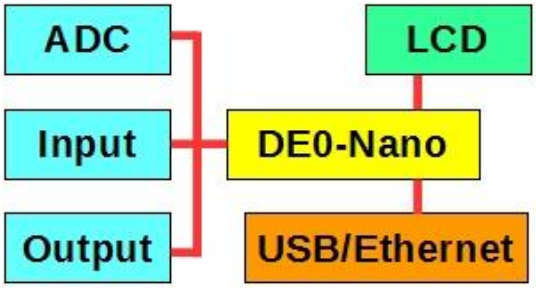
Obrázek 98 - Datový model pro NanoMod.

Zařízení NanoMod bude komunikovat pomocí paketů uvedených v kapitole 9.3. Datový model adresního prostoru je mírně upraven dle požadavků na naše zařízení, viz obr. 98.

10.1 HARDWARE NANOMOD

Základem zařízení je vývojový kit DE0-Nano od firmy Terasic Inc. obsahující kromě vestavěného programátoru USB-Blaster také dvě 40pinové lišty pro připojení dalších obvodů.

Pro vlastní realizaci zařízení bylo nutné navrhnout, osadit a oživit přídavné desky sloužící buďto jako konvertory rozhraní, galvanické oddělení signálů či zobrazovač stavových údajů. Galvanické oddělení signálů slouží především jako ochrana zařízení, ale i také umožňuje pracovat s 24V logikou klasických PLC. Fotografie celého funkčního zařízení je uvedena v příloze. Zařízení je doplněno o USB Hub sloužící pro napájení komponent a routeru hAP lite od Mikrotiku. Router poskytuje připojení NanoModu jak prostřednictvím klasického ethernetu tak i prostřednictvím WiFi.



Obrázek 99 - Blokové schéma zařízení NanoMod.

Pro návrh desek PS byl použit softwarový nástroj Eagle PCB ve volně šiřitelné verzi (omezení velikosti desky DPS). Pro galvanické oddělení digitálních vstupů byly použity čtyřnásobné optočleny K845P. U digitálních výstupů byla použita elektromechanická relé S1A050D. Při návrhu galvanického oddělení analogových vstupů byly použity optoizolační zesilovače HCPL-7800 a DC-DC převodník napětí. U optoizolačních zesilovačů, jak již název napovídá, je použito optické oddělení signálu za pomocí ∑Δ modulace. Princip modulace vychází z principu, že se přenáší vždy rozdíl mezi současným a předcházejícím vzorkem, tj. rozdíl je přenášen pomocí jediného bitu přes integrovaný optočlen.

10.2 FIRMWARE NANOMOD

Firmware je pro jednotku napsán v programovacím jazyku C s využitím vývojového prostředí Eclipse. Skládá se z hlavního programu, kde po spuštění je provedena základní inicializace a konfigurace systému. IP adresa zařízení je nastavena staticky na hodnotu 192.168.88.5. Následuje vypsání základních informací o systému na LCD displej. Na poslední řádek LCD displeje je vypisován vždy poslední obdržený typ Modbus příkazu, např. *Write Multiple Coils*. Dále následuje nekonečná smyčka *while* obsahující příkaz switch pro detekci jednotlivých Modbus příkazů uvedených v kapitole 9.3.

Pro otestování firmware je ve skriptovacím jazyce Python vytvořeno několik úloh pojmenovaných podle testovaného Modbus příkazu:

- ReadCoils.py – čtení digitálních vstupů,
- ReadInputRegisters.py – čtení ADC,
- WriteMultipleCoils.py – nastavení digitálních výstupů,
- WriteSingleCoil.py – nastavení digitálního výstupu.

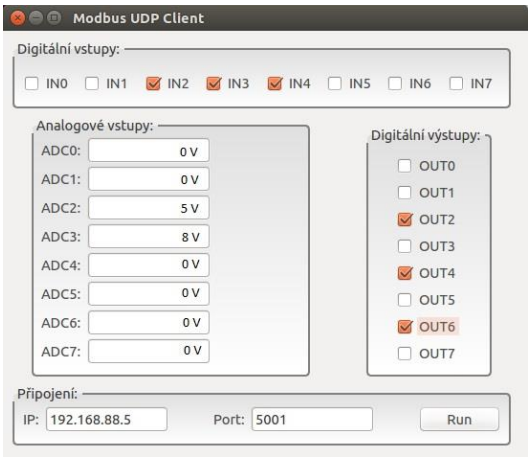
NanoMod obdrží paket ve tvaru uvedeném na obrázku č. 91. Z prvních 7 bajtů tvořících MBAP hlavičku použijeme pouze 5. a 6. byte, který udává délku PDU včetně UID. PDU data začínají 8. bajtem, který udává typ Modbus příkazu. Proto jsou nepotřebná data odstraněna a následně volána funkce *ctiModbus*. Po zpracování přijatého příkazu je vždy zaslána odpověď nadřazenému PLC.

```
…
 //precti data
 Socket0_Recvfrom(UDP_data);

 //odstran MBAP a zjisti delku PDU a typ zpravy
 delka_pdu=data[5]+data[4]*256-1;
 memcpy(data,data[7],delka_pdu);
 typ=data[0];

 //proved MODBUS prikaz
 res=ctiModbus(data);

 //posli odpoved
 if(res==0)
 {
   res=Socket0_Sendto(data,strlen(data));
 }
 else {
   data[0]=data[0] | 0x80;
   data[1]=0x01;
   res=Socket0_Sendto(data,2);
}
```

Obrázek 100 - Ukázka části zdrojového kódu.

10.3 APLIKACE PRO NANOMOD

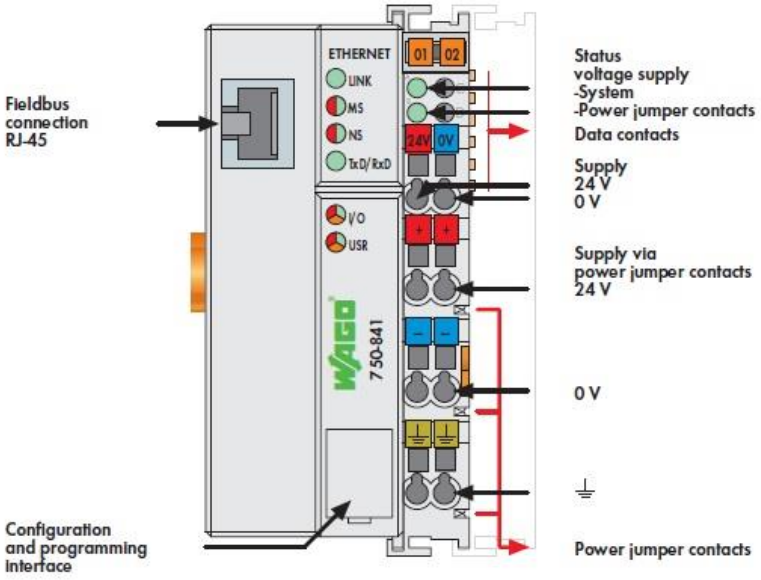
Pro důkladné otestování celého zařízení byla vytvořena grafická aplikace v jazyce Python za použití Qt designeru. Postup tvorby aplikace za použití Qt designeru byl podrobně popsán v kapitole 5.3. Program po stisku tlačítka *Connect* periodicky komunikuje se zařízením Nanomod pomocí protokolu Modbus TCP. Program lze ukončit stisknutím tlačítka *Disconnect*. Uživatel může pouze ovlivňovat stavy digitálních výstupů, ostatní hodnoty jsou vyčteny z NanoMod, tj. korespondují se skutečným stavem.



Obrázek 101 - Testovací aplikace. Soubor modbus_app.py.

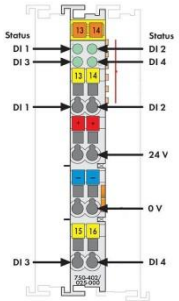
11 WAGO

Modulární řídící systém firmy WAGO řady 750-841 využívá v základu procesorový modul s 32-bitovým procesorem Intel StrongARM s rozhraním Ethernet (10/100 Mbit/s). K tomuto modulu je pak k dispozici řada rozšiřujících modulů, ať již analogových či digitálních vstupů/výstupu, tak i širokou řadou komunikačních rozhraní.



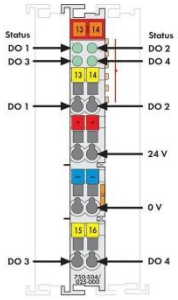
Obrázek 102 - Popis PLC Wago. Zdroj [14].

Pro otestování modulu NanoMod je procesorový modul rozšířen o dva moduly. První z nich je číslicový vstupní modul 750-402 obsahující čtyři vstupní kanály s napěťovou úrovní 24 V DC, které se adresují po bitech (viz PLC configuration). Status vstupních kanálů je indikován pomocí LED diod.



Obrázek 103 - Vstupní modul 750-402. Zdroj [14].

Druhý je číslicový výstupní modul 750-504 obsahující čtyři výstupní kanály s napěťovou úrovní 24 V DC, které se také adresují po bitech. Status výstupních kanálů je opět indikován pomocí LED diod. Všechny popisované moduly jsou určeny pro montáž na DIN lištu. Cena jednoho v/v modulu se pohybuje od cca 2000 Kč.



Obrázek 104 - Výstupní modul 750-504. Zdroj [14].

11.1 CODESYS

CoDeSys je univerzální vývojové prostředí pro aplikační programy řídicích systémů PLC. Bylo vytvořeno firmou 3S podle standardu IEC 61131-3. Bez ohledu na použitý hardware je tak možno vytvářet jednotné datové a programové struktury a způsoby ovládání. Toto vývojové prostředí využívá více výrobců programovatelných automatů jako například Wago, Eaton, Berghof či ABB. Prostředí umožňuje kromě programování (textové a grafické editory) také ladění programu nebo vizualizaci procesu. CoDeSys poskytuje různé programovací jazyky, které je možno volit podle charakteru dané úlohy. V rámci jednoho projektu je pak lze libovolně kombinovat.

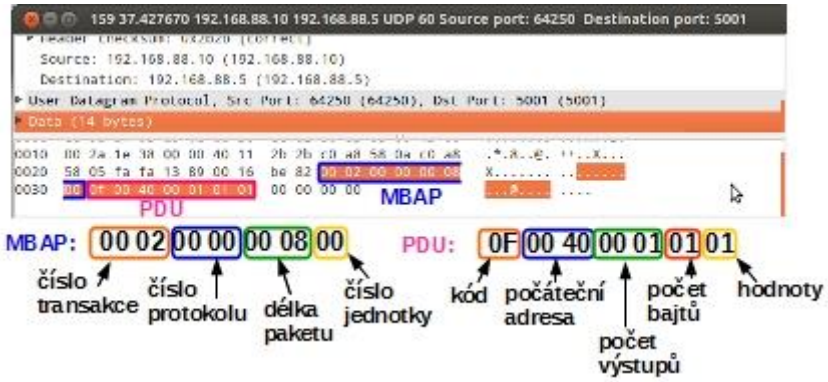
K dispozici jsou tyto jazyky :

- IL - Instruction List,
- ST - Structured Text (obdoba Pascalu),
- LD - Ladder Diagram (obdoba kontaktů a relé),
- SFC - Sequential Function Chart,
- FBD - Function Block Diagram,
- CFC - Continuous Function Chart.

Součástí základní verze prostředí CoDeSys jsou knihovny univerzálních funkcí a funkčních bloků. Každý výrobce pak k systému dodává vlastní knihovny programových funkcí, popř. aplikační úlohy.

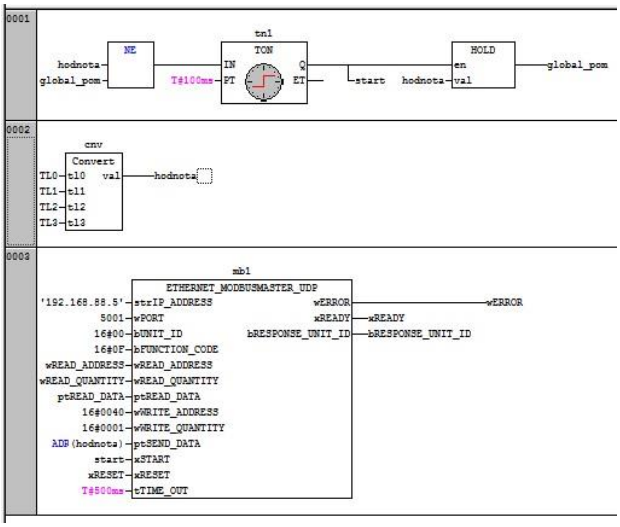
11.2 TESTOVACÍ ÚLOHY

Pro programovatelný komunikační modul 750-841 byly vytvořeny tři úlohy pro otestování činnosti navrženého modulu NanoMod. První úloha po stisku jednoho ze čtyř tlačítek připojených k PLC prostřednictvím modulu 750-402 zajistí vyslání Modbus paketu do modulu NanoMod. Na tomto modulu je pak sepnuto příslušné výstupní relé. Po uvolnění příslušného tlačítka je opět poslán Modbus paket zajišťující rozepnutí příslušného relé. Pro tuto činnost se využívá Modbus příkaz *Write Multiple Coils*, viz obrázek č. 105, znázorňující zachycený paket od PLC.



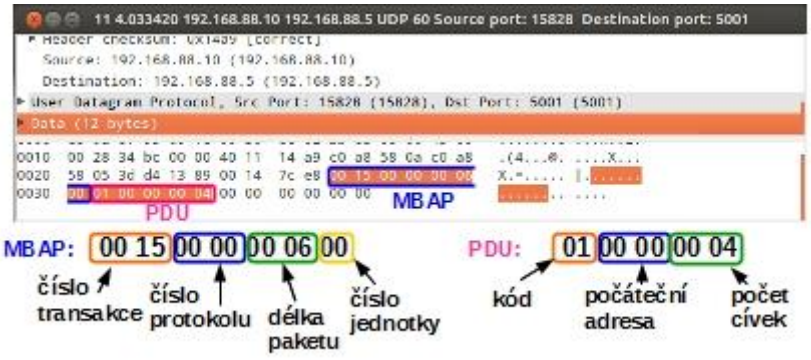
Obrázek 105 - Zachycený paket Write Multiple Coils.

Ze zachyceného paketu je patrná struktura Modbus protokolu, který se skládá z MBAP hlavičky a PDU části. V případě, že jednotka NanoMod je nedostupná, popřípadě, že UDP paket nebyl doručen, PLC sepne signalizaci poruchy (typ chyby - wERROR). Celá aplikace pro PLC Wago je napsána v jazyce FBD (Function Block Diagram) pomocí blokového schématu, viz obrázek č. 106. Aplikace se skládá ze tří částí, první část v případě změny obsahu proměnné *hodnota* zajišťuje vygenerování startovacího impulsu pro Modbus komponentu. Druhá část slouží pouze pro převod tlačítek do binární podoby. Ve třetí části je použita zmíněná komponenta pro komunikaci protokolem Modbus.



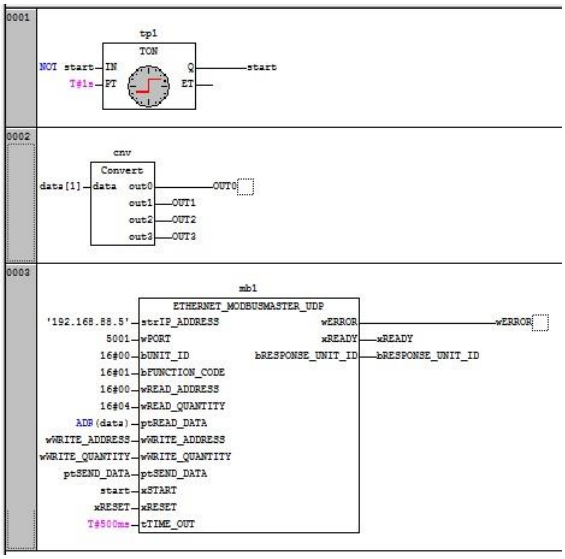
Obrázek 106 - Blokové schéma FBD, první úloha.

Druhá úloha pracuje obráceně, periodicky je zde dotazován modul NanoMod a z něho jsou získávána data ze vstupních kontaktů. Hodnoty těchto vstupů pak kopírují výstupní relé PLC. Pro tuto činnost byl použit rozšiřující modul 750-504. Opět pro jednoduchost je PLC aplikace napsána v jazyce FBD. UDP paket zobrazující Modbus příkaz, které vysílá PLC Wago, znázorňuje obrázek č. 107.



Obrázek 107 - Zachycený paket Read Coils.

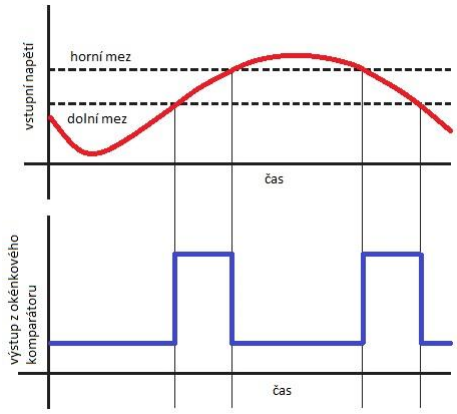
Aplikace se skládá ze tří částí, první část vytváří astabilní klopný obvod s periodou 1 sekunda. V druhé části se provádí dekódování binárního čísla na jednotlivé bity a třetí část zůstává shodná, pouze jsou změněny parametry.



Obrázek 108 - Blokové schéma FBD, druhá úloha.

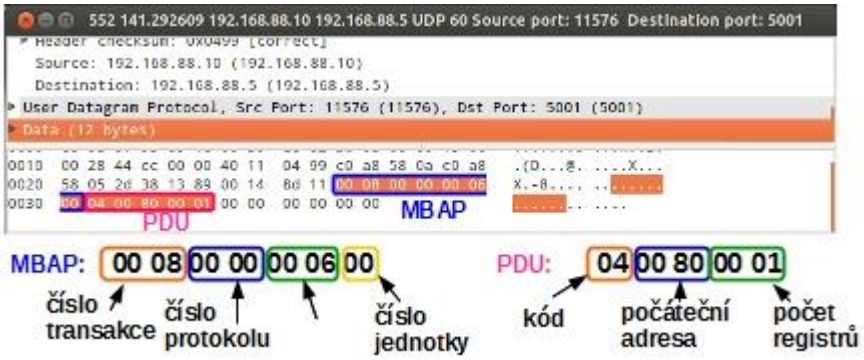
Komponenta Ethernet_ModbusMaster_UDP ukládá získaná data do pole typu *WORD*, ze kterého pak pomocí logických součinů získáme původní logické hodnoty (hodnoty vstupních tlačítek). Vzhledem ke zvolenému způsobu komunikace master-slave zde dochází k určitému zpoždění reakce na změnu vstupu. V případě požadavku na rychlou reakci při změně stavu na vstupech, by se modul NanoMod musel choval jako client a PLC Wago jako Modbus server, tj. role by si vzájemně vyměnily.

Třetí úloha je zaměřena na měření analogových hodnot a jejich zpracování, přesněji řečeno na vytvoření programovatelného okénkového komparátoru. Pro komunikaci s NanoMod modulem použijeme stejný princip komunikace jako v druhé úloze, změníme však typ Modbus příkazu.



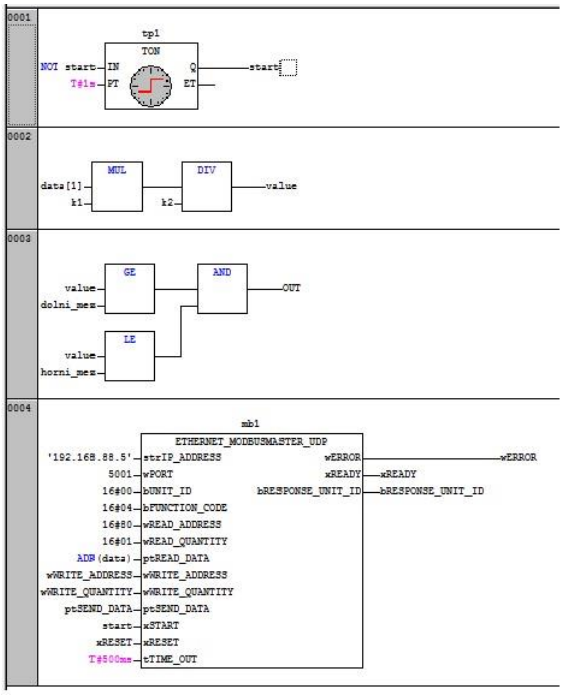
Obrázek 109 - Průběh okénkového komparátoru.

Naměřená hodnota je pak porovnávána s dvěma požadovanými hodnotami (tzv. regulační okno). V případě, že naměřená veličina leží v rozmezí těchto hodnot, bude výstupní signál *out* nabývat log. 1 (viz obrázek. č. 109), respektive bude výstupní relé sepnuto. V ostatních případech bude výstupní relé v rozepnutém stavu.



Obrázek 110 - Zachycený paket Read Input Registers.

Aplikace se skládá ze čtyř částí, první a čtvrtá zůstává téměř shodná. Ve druhé části se provádí převod binární hodnoty získané z AD převodníku na skutečnou hodnotu napětí, neboť na vstupu ADC je implementovaný dělič napětí. Ve třetím bloku je realizován programovatelný okénkový komparátor. Hodnoty mezí lze libovolně měnit, čímž získáme zmíněnou programovatelnost.



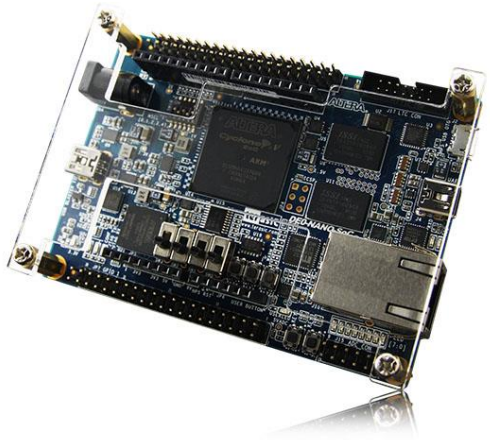
Obrázek 111 - Blokové schéma FBD, třetí úloha.

12 ZÁVĚR

Při návrhu tohoto zařízení jsem si prohloubil své znalosti v oblasti programovatelných hradlových polí, zejména v realizaci a programování soft-procesoru Nios. Dále jsem si rozšířil znalosti v oblasti automatizace, především v komunikaci PLC s externími zařízeními.

Navrhnuté zařízení je použitelné v širokém spektru aplikací v oblasti průmyslové automatizace, měření a regulace, ve kterých je potřeba rozšířit PLC o další vstupy a výstupy, popřípadě realizovat speciální komunikační rozhraní či komunikační protokol. Dále lze zařízení využít samostatně v návaznosti na softwarovou aplikaci realizující jednoduchý řídící systém, popř. vizualizaci procesu.

V současné době uvažuji o návrhu a realizaci kompaktního PLC s upevněním na DIN lištu na bázi kitu DE0-Nano-SoC od firmy Terasic Technologies Inc. Jako programovací jazyk bych použil skriptovací jazyk Python, který by sloužil k řízení, podobně jako jazyk ST (jazyk strukturovaného textu) u klasického PLC. Toto řešení nabízí podstatně větší variabilitu návrhu, odpadá zde nutnost řešit komunikační protokol mezi moduly a snižuje náklady na realizaci zařízení na minimum. V případě potřeby také umožňuje přímou komunikaci s databázemi, popř. s dalšími systémy. Podobným směrem se vydala i společnost Beckhoff se svým produktem TwinCAT 3 (integrován do Microsoft Visual Studia).



Obrázek 112 - Kit DE0-Nano-SoC . Zdroj [10].

Z mého pohledu je zadání diplomové práce splněno a umožňuje použití modulu v praxi. Modul je napájen přes 5V napájecí USB adaptér a má všechny vstupy a výstupy galvanicky oddělené. S řídícím PLC systémem komunikuje přes rozhraní Ethernet nebo WiFi. Celé zařízení je navrhnuto za použití volně

dostupných vývojových nástrojů a je připraveno pro případnou malosériovou výrobu. Cena kompletního zařízení je cca 8 tisíc korun, což je vzhledem k jeho variabilitě cena přiměřená.

POUŽITÁ LITERATURA

[1] SUMMERFIELD, Mark. *Python 3: výukový kurz*. vyd. 1. Brno: Computer Press, 2010. ISBN 978-80-251-2737-7.

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

[3] ŠMEJKAL, Ladislav a Marie MARTINÁSKOVÁ. *PLC a automatizace*. 1. vyd. Praha: BEN - technická literatura, 1999. ISBN 80-86056-58-9.

[4] ŠMEJKAL, Ladislav. *PLC a automatizace 2*. 1. vyd. Praha: BEN - technická literatura, 2005. ISBN 80-7300-087-3.

[5] *Pandatron.CZ: Elektrotechnický magazín* [online]. ©2000-2016 [cit. 2016-03-15]. Dostupné z: http://pandatron.cz/

[6] *ALTERA: now part of Intel* [online]. ©1995-2016 [cit. 2016-03-15]. Dostupné z: https://www.altera.com/

[7] *MICROCHIP* [online]. ©1998-2016 [cit. 2016-03-15]. Dostupné z: http://www.microchip.com/

[8] *HW.cz*: *Vše o elektronice a programování* [online]. c2016 [cit. 2016-03-15]. Dostupné z: http://www.hw.cz

[9] *Wikipedie*: *otevřená encyklopedie* [online]. c2016 [cit. 2016-03-15]. Dostupné z: http://cs.wikipedia.org

[10] *Terasic Technologies: Expertise in FPGA/ASIC Design* [online]. ©2003-2013 [cit. 2016-03-15]. Dostupné z: http://www.terasic.com.tw/en/

[11] *EAGLE online* [online]. ©1994-2016 [cit. 2016-03-15]. Dostupné z: http://www.eagle.cz/

[12] *Regulační pohony: a jejich komponenty* [online]. 2016 [cit. 2016-03-15]. Dostupné z: http://www.regulacni-pohony.cz/

[13] *Dangerous Prototypes: A new open source hardware project every month* [online]. c2011 [cit. 2016-03-15]. Dostupné z: http://http://dangerousprototypes.com/

[14] *WAGO* [online]. 2016 [cit. 2016-03-24]. Dostupné z: http://www.wago.cz/

[15] *Západočeská univerzita v Plzni* [online]. ©1991-2016 [cit. 2016-03-24]. Dostupné z: http://www.zcu.cz/

[16] *Verilog Tutorial* [online]. ©1998-2014 [cit. 2016-03-24]. Dostupné z: http://www.asic-world.com/verilog/veritut.html

Přílohy

Diplomová práce, všechny zdrojové kódy, schémata a návrhy DPS, fotografie, datasheety jsou k dispozici v elektronické podobě na přiloženém CD.