



ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra elektrických pohonů a trakce

Možnosti využití SoC platformy procesorů pro řízení elektrických pohonů

Possibilities of Using SoC Platform Processors for Controlling Electric Drives

Diplomová práce

Studijní program: Elektrotechnika, Energetika a Management

Studijní obor: Elektrické pohony

Vedoucí práce: doc. Ing. Jan Bauer, Ph.D.

**Petr Zakopal
Praha 2023**

PROHLÁŠENÍ

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne _____

Petr Zákopal

PODĚKOVÁNÍ

Tímto bych rád poděkoval vedoucímu této práce doc. Ing. Janu Bauerovi, Ph.D. za skvělé vedení práce a cenné rady při jejím vytváření. Dále bych rád poděkoval všem, kteří mě v mé dosavadním studiu podporovali.

ABSTRAKT

Cílem této práce je prozkoumat možnosti využití heterogenních platform SoC s FPGA pro realizaci výpočtů v reálném čase, zejména pro řízení elektrických pohonů a pro analýzu výkonových prvků pomocí HIL. V textu jsou prezentovány postupy, které vedou k úspěšnému vytvoření FPGA akcelerované aplikace, realizované na vývojové desce Xilinx Kria KR260 s K26 SOM. K vytvoření akcelerované aplikace byly využity SW nástroje PetaLinux, Vivado a Vitis IDE. V rámci práce byla vytvořena aplikace simulující řízení asynchronního motoru pomocí FOC. Zvolené řešení akcelerované aplikace neposkytuje uspokojující výsledky ohledně rychlosti zpracování dat v reálném čase. V závěru práce je navrhováno řešení, které by v případě realizace mohlo splnit daná časová omezení. Na základě vytvořené práce je možné získat přehled o postupu tvorby akcelerované aplikace na platformě firmy Xilinx, Inc.

Klíčová slova: SoC, MPSoC, Xilinx, Kria KR26, Kria K26, ZynQ, FPGA, heterogenní systém, HIL, Hardware in the loop, hardwarová akcelerace, PetaLinux, Vitis, Vivado, HLS, High Level Synthesis, RTL, řízení elektrických pohonů, kernel, Real Time Linux Patch, SPI, FOC, simulace, Field Oriented Control.

ABSTRACT

The goal of this thesis is to research the possibilities of using heterogeneous platforms consisting of SoC with FPGA for making real-time calculations, which could be utilized for controlling electric drives or/and analysing the behaviour of power electronic parts. The text describes the flow of creating the accelerated applications on the Xilinx Kria KR260 development board which utilizes the K26 SOM. The PetaLinux, Vivado and Vitis IDE tools were used for creating the FPGA accelerated application. The possibilities were demonstrated by an application which simulates the FOC of an induction motor. The used solution of accelerated application does success in maintaining the time constraints needed for real-time calculations. However, at the end of the thesis the author presents creating another solution, which could be utilised to meet the given time constraints for real-time applications. By following the text, the reader can gain insight on how to develop a basic accelerated application utilizing Xilinx, Inc. development board.

Keywords: SoC, MPSoC, Xilinx, Kria KR26, Kria K26, ZynQ, FPGA, heterogeneous system, HIL, Hardware in the loop, hardware acceleration, PetaLinux, Vitis, Vivado, HLS, High Level Synthesis, RTL, electric drives control, kernel, Real Time Linux Patch, SPI, FOC, simulation, Field Oriented Control.

OBSAH

1	Úvod.....	1
2	System on a chip.....	2
2.1	Application Specific Integrated Circuit	2
2.2	Aplikace SoC	2
3	System on Modules	3
3.0.1	Embedded Systems.....	3
3.0.2	Hardware Accelerated Applications	4
3.0.3	Výpočetní technika, mobilní zařízení a elektronika	5
4	Programovatelné hradlové pole – FPGA	6
4.1	Vývoj FPGA z PLD	6
4.2	Aktuální složení FPGA	6
4.2.1	Generátory funkcí	7
4.2.2	Paměťové elementy	8
4.2.3	Logické buňky	8
4.2.4	Logické bloky	9
4.2.5	Propojení bloků.....	9
4.2.6	I/O bloky	9
4.2.7	Bloky speciálních funkcí	9
4.3	Programování	9
4.3.1	Forma tvorby algoritmu pro FPGA	10
4.3.2	Konverze HDL na konfigurační bitstream	10
4.4	Spotřeba	11
4.5	Využití	11
4.5.1	Aplikace v nepohonářských odvětví	11
4.5.2	Aplikace v elektrických pohonech.....	12
5	Vývojová deska Digilent Zybo	13
5.1	Základní přehled	13
5.1.1	CPU a FPGA čip	13
5.1.2	Uspořádání vývojové desky Zybo Zynq-7000	15
6	Vývojová deska Xilinx Kria KR260.....	18
6.1	Základní přehled	18
6.1.1	CPU a FPGA čip	18
6.2	Uspořádání vývojové desky	20
6.2.1	Dostupné K26 SOM	22
7	Porovnání představených SoC/SoM platforem pro řízení elektrických pohonů	23
7.1	Konektivita	23
7.2	PS a PL	23

7.3	Developer Experience	24
7.4	Aplikace a operační systém	25
8	Zpětnovazební vektorová regulace	26
9	Model stroje	28
9.1	Matematický popis „kompletního“ modelu stroje	28
9.2	I-n model asynchronního motoru	29
10	Použité nástroje pro vývoj aplikace pro PS a PL	31
10.1	Xilinx Vivado	31
10.2	Xilinx Vitis	31
10.3	PetaLinux Tools	32
10.4	RealTime Linux Patch	33
10.4.1	Postup aplikace PREEMPT_RT patch	34
10.5	Programovací prostředí – operační systém Linux	35
11	Struktura složek	36
12	Tvorba HW architektury Xilinx Vivado	37
12.1	Vivado Board Files	37
12.2	Tvorba HW designu pro Xilinx Kria KR260 vývojovou desku	37
12.2.1	Konfigurace PS pro využití implementovaných periferií	46
12.2.2	Design Constraints	50
12.2.3	HW block design vyvíjené aplikace	51
13	Tvorba PetaLinux	53
13.1	Hardware konfigurace PetaLinux	53
13.2	Konfigurace jádra PetaLinux	54
13.2.1	Konfigurace Device Tree	55
13.3	Konfigurace Root File System	56
13.4	Závěrečný build PetaLinux, generování SDK a tvoření WIC obrazu systému	58
13.5	Spuštění systému PetaLinux na KR260	59
13.6	Nastavení Ethernet adaptéru po spuštění systému	59
14	Tvorba akcelerované aplikace ve Vitis IDE	61
14.1	Tvorba platformy pro akcelerovanou aplikaci	61
14.2	Tvorba application project	62
15	Deployment aplikace na platformu	65
16	Popis pracoviště	67
17	Vytvořená aplikace	69
17.1	Bezpečnost při uživatelském ukončení aplikace	69
17.2	Keyboard Input	71

17.2.1	Konfigurace PL	71
17.2.2	PS Aplikace	71
17.2.3	Interakce s kernelem	72
17.3	CPU/FPGA Model	75
17.3.1	Zobrazení výsledků simulace	77
17.4	Preloaded Data	77
17.5	SPI	79
17.5.1	Zapojení pro testování SPI komunikace	82
17.6	Timer Thread	84
17.6.1	Thread Loop	84
17.7	Akcelerované algoritmy (kernely) v PL	90
17.7.1	Implementace výpočtu diferenciálních rovnic	91
17.7.2	Implementace regulátorů	91
17.7.3	Implementace modulace prostorového vektoru	92
17.7.4	Implementace modelu invertoru	93
17.7.5	Implementace modelu asynchronního motoru	93
18	Poznatky získané profilováním aplikací	94
18.1	Preloaded Data – Legacy Application	94
18.2	CPU/FPGA	97
18.3	Využití zdrojů pro PL jednotlivých akcelerovaných aplikací	99
	Závěr	100
	Literatura	105
Příloha A	Seznam symbolů a zkratek	106
A.1	Seznam zkratek	106
A.2	Seznam symbolů	110
Příloha B	Tvorba HW designu pro Digilent Zybo Zynq-7000	113
Příloha C	Upravený postup nahrávání aplikace pro Digilent Zybo	122
Příloha D	Python skript pro vykreslení výsledných průběhů simulace	123

SEZNAM OBRÁZKŮ

3 - 1	Blokové schéma embedded systému a řízeného fyzikálního systému. (převzato z [8], upraveno)	4
4 - 1	Blokové schéma složení moderních FPGA.....	6
4 - 2	Základní koncept uspořádání FPGA.....	7
4 - 3	Ukázka, jakým způsobem realizuje funkční generátor požadovanou funkci pomocí SRAM a MUX. (inspirováno [7])	8
4 - 4	Blokové schéma převodu aplikace naprogramované v procedurálním jazyce na bitstream, kterým je konfigurováno FPGA.	11
5 - 1	Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – boční pohled.	13
5 - 2	Detailní schéma čipu Zynq-7000, umístěného na vývojové desce <i>Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board</i> . (převzato z [23])	15
5 - 3	Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – vrchní pohled s vyznačením komponent.	15
5 - 4	Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – spodní pohled.	16
6 - 1	Vývojová deska Xilinx Kria KR260 – boční pohled.....	18
6 - 2	Blokový diagram K26 SOM Kria. [5]	19
6 - 3	Vývojová deska Xilinx Kria KR260 vrchní pohled s vyznačením komponent.	21
6 - 4	Vývojová deska Xilinx Kria KR260 – spodní pohled.	21
8 - 1	Obecné schéma zpětnovazební vektorové regulace. (převzato z [33], [34], upraveno)....	27
8 - 2	Simulační schéma zpětnovazební vektorové regulace, realizované v této práci. (převzato z [33], [34], upraveno)	27
10 - 1	Blokový diagram tvorby spustitelné aplikace v prostředí Vitis. (převzato z [27], upraveno)	32
10 - 2	Graf prováděné simulace při testování PREEMPT_RT Linux Patch.	33
12 - 1	Xilinx Vivado – volba typu projektu pro Xilinx KR26, použitelného dále jako platforma ve Vitis.	38
12 - 2	Xilinx Vivado – výběr základní komponenty, pro který bude HW design vytvářen.	38
12 - 3	Xilinx Vivado – nabídka vytváření block design.	39
12 - 4	Xilinx Vivado – vložení PS IP bloku.	40
12 - 5	Xilinx Vivado – automatické propojení pro PS.	40
12 - 6	Xilinx Vivado – zablokování FPD a odblokování LPD pro block design.	41
12 - 7	Xilinx Vivado – nastavení bloku Clocking wizard.	42
12 - 8	Xilinx Vivado – blokový design s PS a blokem Clocking wizard po provedení automatizace.	42
12 - 9	Xilinx Vivado – propojení bloků Clocking wizard a Processor System Reset.	43
12 - 10	Xilinx Vivado – nastavení bloku AXI Interrupt Controller.....	43
12 - 11	Xilinx Vivado – block design po automatizaci a propojení bloku AXI Interrupt Controller.	44
12 - 12	Xilinx Vivado – nastavení bloku Slice pro odstranění dvou horních bitů signálu.	45
12 - 13	Xilinx Vivado – minimální funkční design s výstupním pinem pro řízení ventilátoru.	45
12 - 14	Xilinx Vivado – záložka nastavení CLK signálů na platformě.	46
12 - 15	Xilinx Vivado – PS I/O Configuration QSPI zařízení.	47

12 - 16	Xilinx Vivado – PS I/O Configuration I2C zařízení	47
12 - 17	Xilinx Vivado – PS I/O Configuration PMU (GPIO 4 a GPIO 5 není vybráno).	47
12 - 18	Xilinx Vivado – PS I/O Configuration SPI zařízení	48
12 - 19	Xilinx Vivado – PS I/O Configuration UART zařízení.....	48
12 - 20	Xilinx Vivado – PS I/O Configuration GPIO zařízení.....	48
12 - 21	Xilinx Vivado – PS I/O Configuration System Watchdog Timers (SWDT).....	48
12 - 22	Xilinx Vivado – PS I/O Configuration Triple Timer Counter (TTC), TTC 0 výstup Wa- veout je využit pro PIN řídící chladící ventilátor na SOM.	48
12 - 23	Xilinx Vivado – PS I/O Configuration USB.	49
12 - 24	Xilinx Vivado – PS I/O Configuration DisplayPort.	49
12 - 25	Xilinx Vivado – PS UltraScale+ Block Design v PS IP.	49
12 - 26	Xilinx Vivado – okno tvorby/vložení Constraints File.	50
12 - 27	Xilinx Vivado – HW block design vyvýjené aplikace.....	52
13 - 1	Xilinx Vivado – snímek obrazovky konfigurace PetaLinux pomocí „petalinux-config“ příkazu.	54
14 - 1	Xilinx Vitis IDE – tvorba platformy pro akcelerovanou aplikaci.	62
14 - 2	Xilinx Vitis IDE – build proces platfromy.	62
14 - 3	Xilinx Vitis IDE – výběr platformy pro vytvoření Application Project.....	63
16 - 1	Blokové schéma uspořádání vývojového pracoviště.....	68
17 - 1	Základní větvení ukázkové aplikace.	69
17 - 2	Keyboard Input větev aplikace – manuální nastavování vstupních hodnot do <i>I-n</i> modelu.	72
17 - 3	Snímek obrazovky konzole, zobrazující postup spuštění podaplikace Keyboard Input a výpis první iterace výsledků z kernelu.....	74
17 - 4	CPU/FPGA Model větví aplikace – matematický <i>I-n</i> model, regulace, ASM model v FPGA.....	76
17 - 5	Časová závislost velikosti mechanické otáčivé rychlosti Ω a velikosti magnetického toku rotoru ψ_2 . Výsledné hodnoty získané ze simulace akcelerované pomocí kernelu v PL.....	77
17 - 6	Preloaded Data větví aplikace – automatické čtení změrených/simulovaných vstupních hodnot do <i>I-n</i> modelu a jejich hromadné zpracování v kernelu.	79
17 - 7	SPI větví aplikace – manuální odesílání dat přes SPI.	80
17 - 8	Názorné schéma připojení PMOD3 Xilinx Kria KR260 k LED Matici s obvodem MAX7219.	82
17 - 9	Oscilogram znázorňující časové průběhy SPI komunikace – 10 MHz CLK signál a MOSI signál přenášející data 0x10000001.....	83
17 - 10	Oscilogram znázorňující detail časového průběhu SPI komunikace – 10 MHz CLK signál.	83
17 - 11	SPI Timer Thread větví aplikace – AXI Timer Thread s automatickým odesíláním dat. ..	84
17 - 12	SPI Timer Thread detail threadLoop části.....	86
17 - 13	Blokové schéma regulátoru s ošetřením anti-windup jevu pomocí principu clamping.	92
18 - 1	System diagram – Vitis Analyzer pro Preloaded Data aplikaci.	94
18 - 2	Timeline Trace – Vitis Analyzer pro Preloaded Data aplikaci při zpracování 1 M SH. Na obrázku je zobrazen celý životní cyklus aplikace od startu až po ukončení.	96

18 - 3	Timeline Trace – Vitis Analyzer pro Preloaded Data aplikaci při zpracování 1 M SH. Na obrázku je zobrazena část, kdy je spuštěn kernel a aplikace v PS čeká na výsledky jeho kalkulací.....	96
18 - 4	System diagram – Vitis Analyzer pro CPU/FPGA aplikaci.	98
18 - 5	Timeline Trace – Vitis Analyzer pro CPU/FPGA aplikaci při zpracování 1000 SH. Na obrázku je zobrazen celý životní cyklus aplikace od startu až po ukončení.	98
18 - 6	Timeline Trace – Vitis Analyzer pro CPU/FPGA aplikaci při zpracování 1000 SH hodnot. Na obrázku je zobrazena přibližená část, kdy dochází k iteraci spuštění jednotlivých kernelů.	99
B - 1	Xilinx Vivado – volba typu projektu pro Digilent Zybo.	113
B - 2	Xilinx Vivado – výběr základního HW, pro který bude vytvářena architektura pro Digilent Zybo.	114
B - 3	Xilinx Vivado – vytváření Block Design pro Digilent Zybo.	114
B - 4	Xilinx Vivado – vložení bloku ZYNQ7 Processing System pro Digilent Zybo.	115
B - 5	Xilinx Vivado – nastavení výstupních taktovacích signálů pro Digilent Zybo.	116
B - 6	Xilinx Vivado – propojení bloků taktování pro Digilent Zybo.	116
B - 7	Xilinx Vivado – minimální funkční blokový design pro akcelerovanou aplikaci pro Digilent Zybo.	117
B - 8	Xilinx Vivado – signalizace vložených AXI GPIO bloků pro LED, BTN, SW na kartě <i>Board/Zybo/GPIO</i> pro Digilent Zybo.	118
B - 9	Xilinx Vivado – block design s využitím GPIO pro LED, BTN, SW propojených s PL pro Digilent Zybo.	119
B - 10	Uspořádání připojení tlačítek, přepínačů a LED k PS a PL pro vývojovou desku Digilent Zybo. [24]....	119
B - 11	Xilinx Vivado – kritická upozornění, zobrazená po validaci designu, která je možné ignorovat.	120
B - 12	Xilinx Vivado – nastavení provádění úkonů syntézy, implementace a generování bitstreamu, volba použitých výpočetních jader a určení, kde se mají procesy vykonávat pro Digilent Zybo.	121
C - 1	Diagram popisující upravený postup pro nahrávání a spouštění akcelerované aplikace pro Digilent Zybo.	122

SEZNAM TABULEK

4 - 1	Pravdivostní tabulka ukázkové funkce, realizované v generátoru funkcí, umístěném v logickém bloku FPGA.....	8
5 - 1	Popis označených komponent na vývojové desce Digilent Zybo Zynq-7000. (informace a značení převzaty z [24]).....	17
6 - 1	Popis označených komponent na vývojové desce Xilinx Kria KR260. (informace a značení převzaty z [25])	20
6 - 2	Porovnání hlavních parametrů Kria K26 SOM Commercial a Industrial. (informace a značení převzaty z [26]).....	22
9 - 1	Štítkové údaje stroje.	28
9 - 2	Změřené parametry stroje.	28
12 - 1	Ukázka nastavených AXI portů v Xilinx Vivado platformě pro Xilinx Kria KR260.	46
14 - 1	Nastavení cest souborů pro platformu ve Vitis IDE v souboru platform.spr.	61
18 - 1	Porovnání vybraných hodnot běhu kernelu v aplikaci Preloaded Data – Legacy App pro 1 milion a 100 tisíc sad hodnot (SH). (UVH – ukládání a výpis hodnot, BUVH – bez ukládání a výpisu hodnot).....	95
18 - 2	Vybrané hodnoty běhu kernelu v podaplikaci CPU/FPGA.....	97
18 - 3	Využití zdrojů PL pro akcelerované aplikace.	99
B - 1	Ukázka nastavených AXI portů v Xilinx Vivado platformě pro Digilent Zybo.	117

1 Úvod

V době, kdy byla od elektrických pohonů požadována spolehlivost, vysoká účinnost a výpočetně nenáročné, ovšem kvalitní řízení, byly k řízení využívány konvenční digitální signálové procesory (DSP). Postupem času však docházelo ke zjištění, že výkon DSP nemusí být dostatečný, zejména pro aplikace, u kterých je vyžadováno provést velké množství náročných výpočtů za co nejkratší čas. Aktuálně nastupuje éra SoC, které v mnoha případech obsahují heterogenní strukturu řídící procesorové jednotky (CPU) a logického programovatelného pole (FPGA). Implementace hradlových polí přináší nejen v řízení elektrických pohonů zvýšení výpočetního výkonu, ale také snížení energetické náročnosti.

Perspektiva SoC a FPGA je podpořena jejich využíváním i mimo obor elektrických pohonů. Z důvodu vysoké propustnosti FPGA, vysokých výpočetních výkonů a nízké energetické náročnosti jsou využívány v AI, machine learningu, zpracování obrazu a jiných nepohonářských aplikacích.

Náročnější postup designování logiky a algoritmů aplikací v FPGA může představovat problém v jejich implementaci. Aplikace je tvořena postupem, který kladne vysoké nároky na vzdělání a zkušenosti vývojářů. Většina FPGA je designována pomocí jazyků Verilog či VHDL, jejichž použití a filozofie mohou pro softwarově orientované programátory představovat značnou překážku. Proto je v poslední době trend tvořit aplikace pomocí vyšší úrovně syntézy (HLS). Při použití HLS je možné tvořit programy ve vyšších programovacích jazycích jako je například C, C++ či Python. Vytvořený kód je přeložen do úrovně registrů (RTL), která je následně syntetizována, implementována a generována do bitstreamu, který slouží ke konfiguraci cílového FPGA zařízení.

V prvních kapitolách této práce jsou v teoretické úrovni představeny SoC, FPGA a jejich možné využití v pohonářských i nepohonářských aplikacích. Z mnoha dostupných produktů na trhu byly vybrány platformy Xilinx Kria KR260 SOM a Digilent Zybo Zynq-7000. Platformy byly vzájemně porovnány z různých hledisek a pro demonstraci možného využití platformy pro simulaci asynchronního motoru řízeného pomocí FOC byla vybrána platforma Xilinx Kria KR260.

V navazujících kapitolách je čtenářovi představen postup, kterým je možné vytvořit kompletní akcelerovanou aplikaci s použitím nástrojů *PetaLinux*, Vivado a Vitis IDE.

V závěru práce je představena realizace akcelerované aplikace, která je schopna využívat SPI komunikaci s externími zařízeními. Pro SPI komunikaci byl vytvořen hardware (HW) přímo v FPGA, který je ovládán z prostředí (user space) aplikace. Na základě analýzy aplikace bylo vyhodnoceno, zda způsob vybrané realizace je vhodný pro provádění výpočtů v reálném čase.

2 System on a chip

System on a chip (SoC) je struktura, využívající integrování různých prvků systému na jeden čip. Integrace prvků na jeden čip značně snižuje nároky na rozměry nosičů, na kterých jsou tyto SoC umístěny. Místo diskrétních čipů, obstarávající jednotlivé funkce, je využito jednoho čipu s integrovanými prvky, které vykonávají požadované funkce.

Při integrování prvků do jedné polovodičové struktury dochází ke značnému snížení potřebných počtu kovových vodivých spojů, snížení časové náročnosti výroby a zvýšení rychlosti přenosu dat. Proto je SoC upřednostňováno před metalicky spojenými diskrétními prvky, vykonávající dané operace.

Označení SoC může představovat mnoho struktur. Obecné rozdělení, které je možné najeznout v literatuře a veřejných zdrojích, je následující [1]:

- SoC využívající mikrokontroléru (obsahující CPU, RAM, ROM),
- SoC využívající pouze mikroprocesoru (obsahující CPU, možné i GPUs, jádra pro specializované výpočty),
- SoC určené pro specifické aplikace (Application Specific Integrated Circuit – ASIC).

Z uvedených rozdělení jsou pro řízení elektrických pohonů nejvíce využívány SoC s mikrokontrolérem a ASIC.

2.1 Application Specific Integrated Circuit

Významnou část SoC tvoří *Application Specific Integrated Circuits, popř. Hardware* (ASICs, ASHW). Při použití těchto SoC je předpokládáno, že pokud je architektura HW specializovaná přímo na vykonávání jedné aplikace, je vysoká pravděpodobnost, že ji bude vykonávat bezchybně, kvalitně a rychle.

Tyto aplikace jsou využívány v širokém spektru oborů, jako je např. zpracování zvuku, videa, výpočtu. Tyto ASIC mohou vykonávat rychlé výpočty pro matematické modely elektrických strojů, které jsou využívány např. pro HIL.

Než je ASIC určen pro velkoprodukci, je nutné jej navrhnout, vyzkoušet a odladit. K tomu slouží např. logická programovatelná pole (FPGA). Pokud velkoprodukce není z ekonomických důvodů možná, jsou FPGA využívány přímo v produkci. Pomocí nich je vytvořena HW struktura, která by byla přítomna na ASIC.

2.2 Aplikace SoC

Systémy na čipu se pro jejich výpočetní výkon, prostorovou a energetickou efektivnost využívají v mnoha aplikacích. Jedno z nejvýznamnějších využití v oboru elektrických pohonů je v embedded systémech a hardwarově akcelerovaných aplikacích.

3 System on Modules

System on modules (SOMs) je struktura, jejíž hlavní součástí je SoC. SOMs se oproti SoC již dodávají na PCB a kromě SoC mají na PCB umístěny další komponenty, které mohou být pro danou aplikaci vyžadovány. [2]

SOMs se mohou dodávat jako vývojové desky [3], které obsahují kromě SOMs také nosnou desku (tzv. Carrier Card, CC) s dalšími komponenty, jež jsou vhodné pro vývoj standardních aplikací. Pokud zákazník pomocí vývojové desky odladil vyvíjenou aplikaci, může zakoupit samostatný SOM na základní desce (base board, BB) a CC pro danou aplikaci navrhnut tak, aby obsahovala pouze komponenty, které daná aplikace využívá. Tímto způsobem je možné snížit cenu konečného výrobku o komponenty, které byly z CC při návrhu odstraněny pro jejich nevyužití. Ovšem při tomto způsobu realizace dochází ke zvýšení finanční a časové náročnosti vývoje.

Výrobce platformy *Kria KR260 Robotics Starter Kit*, použité v této práci, dodává k produktu rozsáhlou dokumentaci [4], [5], podle které je možné individuální CC sestavit.

Příkladem individuálně vytvořené CC je open source projekt od firmy *Antmicro Ltd.* Tato firma vydala open source design CC pro *K26 SOM*, původně využívané na *Kria KV260*, jež je určen pro akcelerování audiovizuálních aplikací. Individuální design využívá totožný SOMs, jako je použit v této práci, ale rozdílný CC. Dokumentace a vytvořený návrh je dostupný v [6].

3.0.1 Embedded Systems

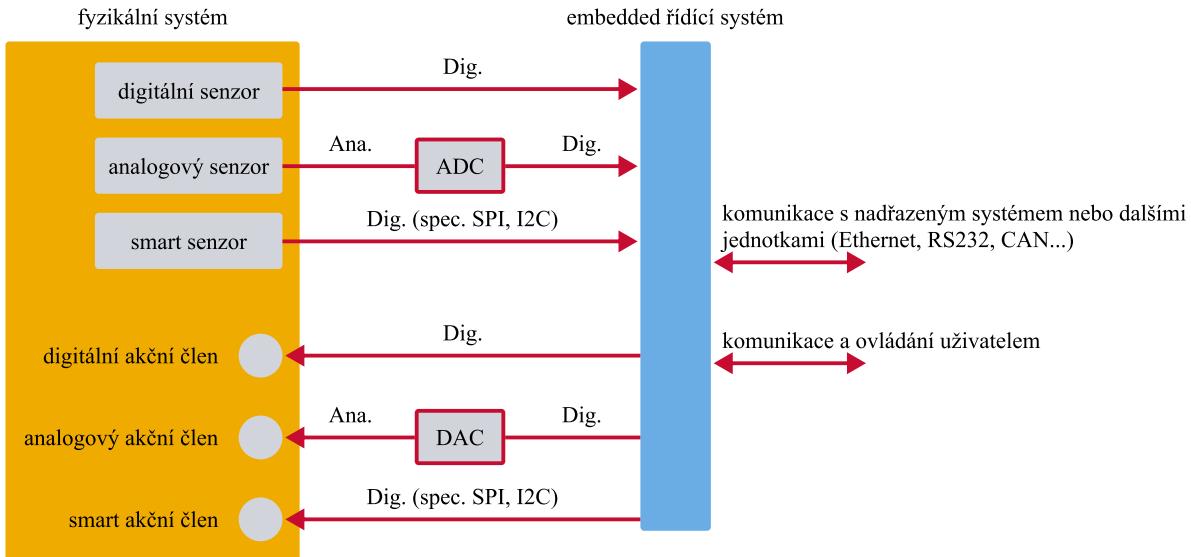
Embedded systems je název pro skupinu prvků, obecně systémů, které je možné charakterizovat jako specifické výpočetní zařízení, resp. počítače, které jsou určeny pro podporu funkce nebo řízení nějakého většího celku, produktu, nebo fyzikálního systému. Naopak osobní počítač je výpočetní zařízení, které je určeno pro vykonávání mnoha univerzálních aplikací a funkcí, které nejsou v okamžiku vývoje počítače určeny, proto nelze mluvit o embedded systému. [7]

Dalším důležitým rozdílem mezi *Embedded System* a obecným výpočetním zařízením je ten, že v případě embedded systému je interakce mezi systémem a uživatelem uměle omezena na základní ovládání či kontrolu funkce. Není předpokládáno, že by uživatel, jež aplikaci embedded systému využívá, výrazným způsobem zasahoval do jeho funkce. Naopak obecný výpočetní systém je uzpůsoben na podstatné zásahy uživatele. [7] [8]

Do embedded systému vstupují signály, které jsou v něm zpracovány a vybrané výsledky výpočtu jsou výstupním produktem systému (v podobě výstupních signálů). Tyto produkty mohou pomocí akčních členů zasahovat do řízeného systému. Vstupní signály jsou většinou získávány ze speciálních snímačů, kompatibilních s embedded systémem (senzor teploty, senzor tlaku, senzor zrychlení, gyroskop, senzory proudu, inkrementální čidla apod.). Jeho výstupní signály jsou například specifická ovládací hodnota napětí, proudu nebo jiné veličiny. Na výstupních pinech mohou být také připojené LED signalizace, komunikační sběrnice některých komunikačních systémů nebo LCD displeje. Způsob, kterým jsou kódovány vstupní a výstupní signály, je většinou specificky určen řízeným systémem. [7]

K obecnému výpočetnímu systému je možné připojit vstupní periferie klasických osbních počítačů (myš, klávesnice, mikrofon). Komunikace embedded systému s periferiemi je většinou standardizována tak, aby bylo možné periferie libovolně zaměňovat bez změny funkčnosti. [7].

Na obrázku 3 - 1 je zobrazeno blokové schéma řízení fyzikálního systému pomocí embedded systému. Tyto bloky mezi sebou komunikují pomocí digitálních signálů. Pokud tyto signály nejsou digitální, musí se před zpracováním v embedded systému zdiskretizovat.



Obr. 3 - 1 Blokové schéma embedded systému a řízeného fyzikálního systému. (převzato z [8], upraveno)

3.0.2 Hardware Accelerated Applications

V mnoha aplikacích je vyžadováno, aby výpočty nebo zpracování dat probíhalo vysokou rychlostí. Tento problém je v některých případech problematické řešit použitím běžného procesoru (CPU), který je optimalizován na provádění obecných komplexních funkcí, řízení běhu uživatelského programu, komunikaci či přesun dat. V moderním světě je třeba zpracovávat množství dat, které v některých případech exponenciálně narůstá. Aby tyto data bylo možné v požadovaném čase s co nejnižším zpožděním zpracovat, je vhodné využít specifický HW a přístup, který bude schopen požadavky rychlosti a výkonu uspokojit. Tento přístup se nazývá *Hardware Acceleration* (hardwarová akcelerace). [9]

Princip hardwarové akcelerace spočívá v přesunu výpočetně náročných aktivit na specifický a oddělený hardware. Celkové řízení běhu aplikace a komunikace je ovšem stále vykonáváno řídícím CPU. Oddělený hardware, na kterém dochází k akceleraci výpočtů, je optimalizován na vykonávanou úlohu a jeho využití přináší zefektivnění běhu celé aplikace. [9]

Struktura, ve které je využíváno více fyzicky oddělených procesorových a hardwarových akceleračních jednotek, se často nazývá heterogenní. [9]

Hardwarová akcelerace je schopna poskytovat rychlejší výpočty než CPU, protože využívá maximální paralelizace výpočtů. Klasické CPU však vykonává jednotlivé instrukce sériově. V případě, že CPU využívá pro výpočty více jader a vláken, je velmi náročné se úrovní paralelizace, při dodržení stejné energetické náročnosti, vyrovnat výpočtům pomocí HW. Pro HW akceleraci je v mnoha oblastech využíváno několik druhů jednotek, které jsou optimalizovány pro dané charakteristiky aplikací.

Graphics Processing Units (GPUs) jsou jednotky, které slouží převážně k akceleraci zpracovávání vizuálních úloh. V době rapidního rozvoje elektroniky a SW je možné využít GPUs v mnoha odvětví umělé inteligence (AI) či kreativních odvětví. GPUs jsou využívány v aplikacích, kde není kladen veliký důraz na nízkou odezvu (latenci). [9]

Tensor Processing Units (TPUs) jsou jednotky, které slouží k provádění algoritmů strojového učení (machine-learning, ML). Jejich přímé datové propojení umožňuje velmi rychlý a přímý přenos dat. Díky přímému připojení nevyžadují využití pamětí, které by přenos dat zpomalovaly. [9]

Field Programmable Gate Arrays (FPGAs) jsou jednotky, ve kterých není při výrobě pevně daná

HW struktura. To umožňuje vytvoření, resp. nakonfigurování HW dle požadavků akcelerované aplikace. FPGAs mohou být využívány i při výpočtech matematických modelů elektrických strojů v reálném čase. Při realizaci této práce je pro akceleraci využíváno právě těchto programovatelných polí.

Vhodným příkladem porovnání časové náročnosti matematických výpočtů pro selektivní eliminaci harmonických složek v trakci pomocí CPU a GPUs je provedeno v [10].

Z článku vyplývá že využitím GPUs skutečně dochází ke snížení potřebného času na provedení výpočtu. V některých případech se jedná o snížení výpočetního času ze 183 ms (při použití CPU) na 0,81 ms (při použití NVIDIA Titan V GPU). Díky využití GPUs je tedy možné algoritmus v lokomotivě provádět v reálném čase.

3.0.3 Výpočetní technika, mobilní zařízení a elektronika

Kromě průmyslových odvětví jsou CPU využívány i pro běžné aplikace spotřební elektroniky.

Protože jsou kladený stále vyšší nároky na výpočetní rychlosť a nižší cenu ve spotřební elektronice, jako jsou mobilní zařízení (mobilní telefony, osobní počítače), servery, začíná převažovat využívání SoC i v těchto oblastech.

Společnost Apple Inc. již téměř ve všech novějších zařízeních používá individuálně navrhnutý SoC. Příkladem je A16 Bionic pro iPhone 14 Pro, Apple M1 a M2 pro tablety a počítače.

Díky specifickým řešením a vylepšeným architekturám (jádra SoC pro vysoký výkon a jádra pro ekonomickou spotřebu energie) bylo možné zvýšit výkon a snížit energetickou náročnost zařízení spotřební elektroniky. [11]

4 Programovatelné hradlové pole – FPGA

4.1 Vývoj FPGA z PLD

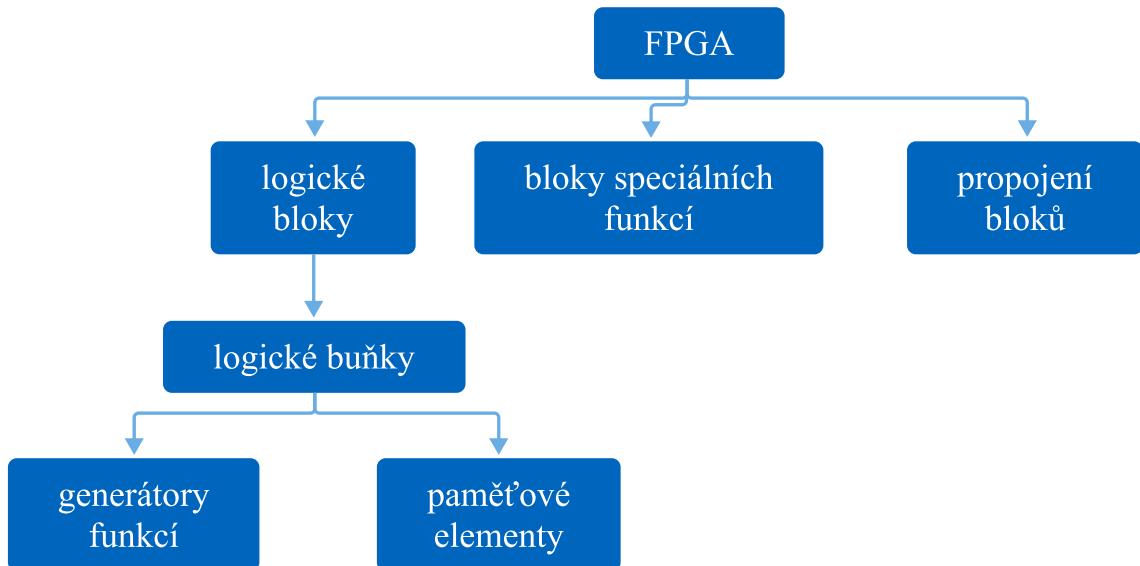
Programovatelná hradlová pole (FPGA) jsou zařízení, jejichž historický vývoj stojí na programovatelných logických zařízení (programmable logic devices, PLD). První PLD fungovala na principu Booleových funkcí součtu násobení (sum of products). Tato zařízení obsahovala matici (proto se také nazývají programmable logic arrays, PLA) více vstupových bloků AND a OR. Programování požadované funkce probíhalo pomocí přerušování vstupů do jednotlivých logických bloků. Později byly do struktury PLA přidány D klopné obvody s multiplexory. Díky těmto součástím bylo možné vytvářet logické kombinační a sekvenční obvody, resp. automaty. Posledním vylepšením PLA, které stálo před zrodem FPGA, spočívalo v umístění více PLA bloků (skládajících se z AND, OR, multiplexeru a D klopného obvodu) na jeden integrovaný čip. Programovatelné spojení různých PLA bloků a výstupů umožnilo vytvořit požadovanou funkci. [7]

4.2 Aktuální složení FPGA

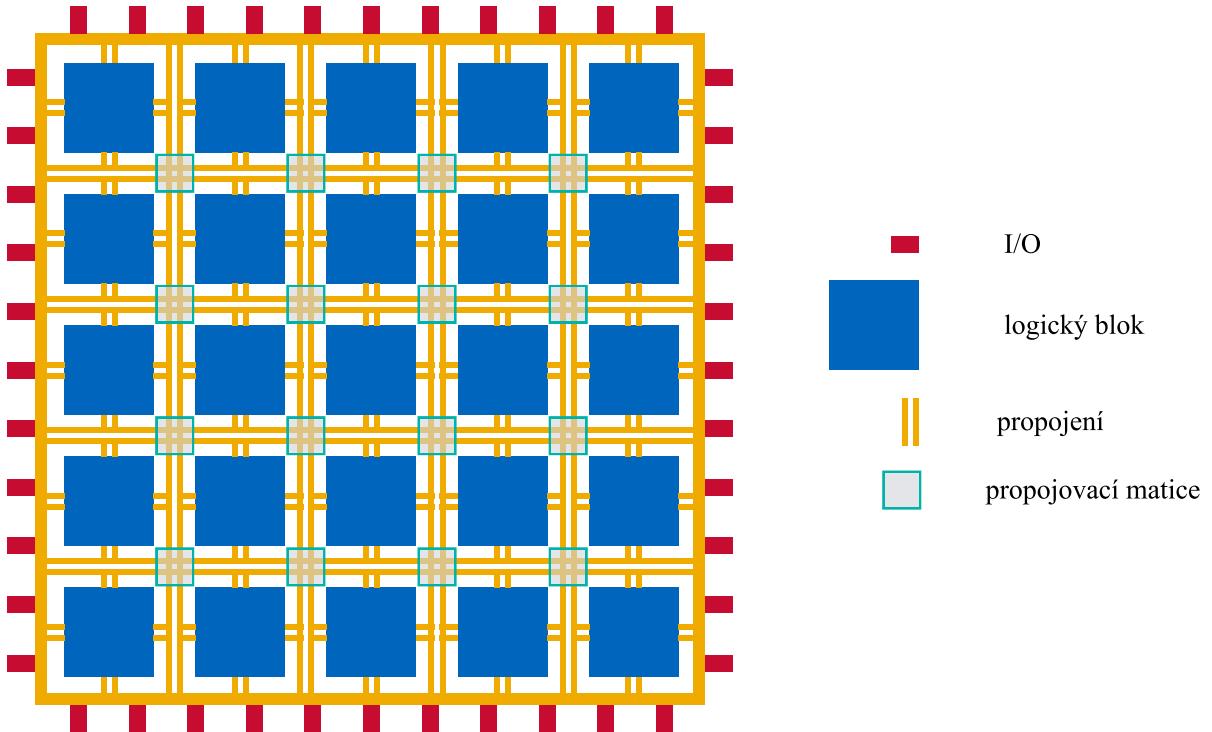
Moderní FPGA se skládají z 2D matice propojených programovatelných logických bloků, bloků speciálních funkcí a propojů. Logické bloky se skládají z mnoha buněk, které se skládají z generátorů funkcí a paměťových elementů. Po obvodě FPGA jsou rozmístěny vstupní a výstupní piny (I/O), připojené na zvláštní logické bloky. [7]

Na obr. 4 - 2 je možné pozorovat schéma základního konceptu uspořádání FPGA. Na schématu jsou vyznačeny logické bloky, jejich propojení, propojovací matice pro aktivování jednotlivých propojů a vstupů a výstupů (I/O) FPGA.

I přesto, že se tato práce převážně věnuje využití SoC a SOMs pro řízení elektrických pohonů je vhodné představit základní části FPGA a nastínit jejich funkci.



Obr. 4 - 1 Blokové schéma složení moderních FPGA.



Obr. 4 - 2 Základní koncept uspořádání FPG. A.

4.2.1 Generátory funkcí

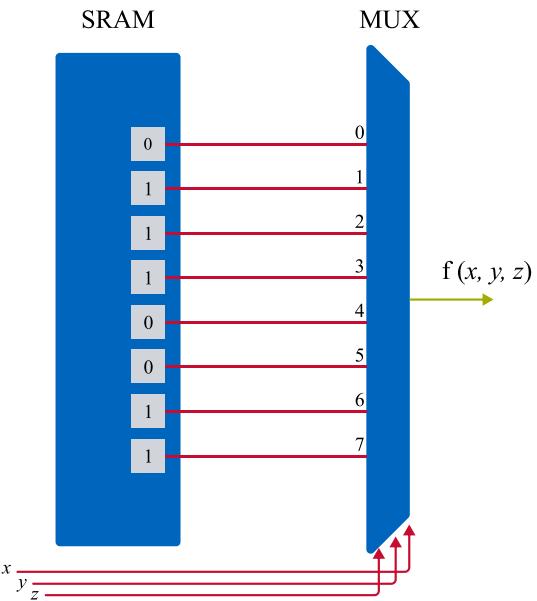
Oproti předchůdcům (PLD), které pro generování funkcí používaly logická hradla tvořená CMOS tranzistory, využívají FPGA tzv. generátory funkcí.

Logickou funkci je možné popsat pravdivostní tabulkou, která má určitý počet vstupů a odpovídající počet výstupů. Dle [7] je možné si představit, že se generátor dané funkce skládá ze samostatné statické paměti (SRAM), jejíž výstupy jsou přímo přivedeny na vstup multiplexeru (MUX). Signály výběru výstupů by odpovídaly vstupním proměnným a jednotlivé vstupy do MUX výstupům funkce.

Pro bližší pochopení generátoru funkcí z předchozího odstavce je možné představit realizaci smyšlené logické funkce $f(x, y, z) = \bar{x}z + y$. Pravdivostní tabulka této smyšlené logické funkce je zobrazena v tab. 4 - 1. Odpovídající realizace pomocí MUX a SRAM je zobrazena na obr. 4 - 3. Tato reprezentace se nazývá look-up table (LUT). Grafické znázornění je inspirováno [7].

Tab. 4 - 1 Pravdivostní tabulka ukázkové funkce, realizované v generátoru funkcií, umístěném v logickém bloku FPGA.

i	x	y	z	$f(x, y, z)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1



Obr. 4 - 3 Ukázka, jakým způsobem realizuje funkční generátor požadovanou funkci pomocí SRAM a MUX. (inspirováno [7])

Výhoda reprezentace funkcí pomocí generátorů funkcí oproti logickým hradlům je, že doba zpoždění signálu (propagation delay) pro funkci je konstantní. Respektivě je konstantní, pokud funkci je možné realizovat jednou LUT. Pro relizace obecné funkce je zapotřebí multiplexer $2^n \rightarrow 1$ a SRAM s počtem buněk 2^n , kde n je počet vstupních proměnných dané funkce. [7]

4.2.2 Paměťové elementy

Paměťové elementy jsou v LUT realizovány pomocí D-klopňových obvodů. Tyto obvody mohou při konfiguraci FPGA být nastaveny tak, že budou reagovat na nástupnou nebo sestupnou hranu časovacího signálu (clock, CLK) řídícího procesoru nebo na úroveň řídícího signálu (latch). [7]

Protože typ latch je citlivý na úroveň signálu, může být problematické dovést požadovaný signál na vstup klopňového obvodu v požadovaném čase. Velmi často jsou proto paměťové členy konfigurovány jako D-klopné obvody reagující na hranu. Pokud je používán signál CLK vyšších frekvencí, je D-klopny obvod reagující na hranu snadněji schopný reagovat v požadovaném čase. [7]

Často jsou na vstup paměťových elementů připojeny výstupy MUX generátorů funkcií. [7]

4.2.3 Logické buňky

Logické buňky jsou elementy, skládající se z generátorů funkcií a paměťových elementů. Velmi často se počet logických buněk údává jako jeden ze základních parametrů FPGA, podle kterého je uživatel možný rozhodnout, zda je obvod vhodný pro jeho aplikaci. Pomocí logické buňky nebo skupiny logických buněk je již možné vytvářet plnohodnotnou kombinační a sekvenční logiku. [7]

4.2.4 Logické bloky

Logické bloky se skládají ze spojení několika *logických buněk* do jedné skupiny. Díky umístění této skupiny buněk na čip geograficky blízko dochází k minimalizaci zpoždění signálu mezi jednotlivými buňkami. Častá skutečnost je, že jednotlivé bloky mohou mít již předkonfigurovanou funkci, jako je např. sčítačka, dělička nebo násobička. [7]

4.2.5 Propojení bloků

Propojení bloků je prováděno z důvodu spojení jednotlivých logických bloků a I/O. Pro spínání určených propojů jsou na čipu mezi jednotlivými propojí umístěny propojovací matice, resp. „přepínače“. Ty slouží ke spojení jinak oddělených propojů, logických bloků a I/O. [7]

Na obr. 4 - 1 je prezentována 2D struktura pole. Ovšem pro zvětšení počtu LUTs a snížení vzdáleností mezi logickými bloky je v moderních FPGA použita 3D struktura, kdy dochází k vrstvení jednotlivých logických bloků a jejich propojů do výšky. [12]

4.2.6 I/O bloky

I/O bloky jsou obvykle umístěny na okraji designu FPGA. Slouží k přivedení resp. vyvedení signálů FPGA na externí připojovací piny struktury. Tyto výstupní bloky mohou využívat různé standardy k přenosu informace typu *single-ended* (napětí vztaženo k referenční nule) (LVTTL, LVCMOS PCI, PCIe, SSTL) nebo typu *double data rate* (diferenciální signál, vztažený k výstupu jiného I/O bloku) (LVDS). I/O bloky jsou strategicky umístěny na okraj struktury, aby byla minimalizována vzdálenost mezi I/O blokem a hranicí FPGA, představující vnější okolí. [7] [12]

4.2.7 Bloky speciálních funkcí

Aby došlo ke zvětšení rychlosti přenosu dat z FPGA do externího CPU a naopak, jsou některé speciální funkce implementovány jako funkční bloky přímo do struktury FPGA. To umožňuje efektivní využití FPGA pro aplikace. [7]

Block RAM (BRAM) je blok, který slouží k uchování dat. Sice by bylo možné vytvořit paměťový blok z *Logických bloků*, ale docházelo by k omezení využití FPGA pro jeho původní aplikaci a pro realizaci bylo potřeba využít mnoho bloků. BRAM mají oddělený vstup a výstup, současně s odděleným CLK. Proto je možné do BRAM zároveň data zapisovat a zároveň z něj číst. [7]

DSP, resp. digital signal processing bloky, slouží ke zpracování digitálního signálu. V těchto blocích jsou implementovány funkce AND, OR, NAND, NOT, násobičky a sčítačky. Mají nízkou spotřebu. DSP bloky jsou často umístěny geograficky blízko bloků BRAM, které slouží jako „mezipaměti“ (buffer). [7]

Procesor, je-li implementovaný do jedné struktury společně s FPGA, dochází ke snížení časového zpoždění při jejich vzájemné komunikaci. [7]

Digital Clock Manager slouží k vytvoření jiného, resp. nižšího taktovacího signálu CLK, který je odvozen z původního vstupního/zdrojového CLK, pro různé bloky v FPGA. [7]

Multi-Gigabit Transcievers slouží k přenosu dat takovým způsobem, aby došlo k minimalizaci vlivu rušení na přenášená data. Obecně obstarávají optimální serializaci a paralelizaci dat. [7]

4.3 Programování

Ve skutečnosti není možné mluvit o programování FPGA. Při tvorbě „programu“ pro FPGA dochází k vytváření struktury HW, jež bude v FPGA vytvořena.

4.3.1 Forma tvorby algoritmu pro FPGA

K programování, resp. konfiguraci FPGA je možné přistupovat z několika úrovní. Jednou z využívaných metod popisu požadovaného HW na FPGA je popis struktury/toku signálu obvody (structural/data flow circuits). K tomuto popisu je využíváno jazyků HDL, VHDL a Verilog (Hardware Description Language, Very High-Speed Integrated Circuit Hardware Description Language, -). V těchto jazycích je využíváno logických členů AND, OR, NOT nebo bloků sčítáček a násobiček. Forma popisu, jež naopak využívá vyššího programovacího jazyka než HDL, je nazývána metoda popisu chování obvodů (behavioral circuits). Zatímco HDL slouží k popisu hardware s využitím nízké míry abstrakce, popis ve vyšších programovacích jazycích, které popis pomocí behavioral circuits umožňuje, je pro programátory (zejména ty softwarové) značně příjemnější, protože využívá běžných procedurálních programovacích jazyků jako je C, C++ nebo Python. Tyto jazyky jsou následně přeloženy/kompilovány do HDL. Po překladu do HDL pomocí *high level synthesis* (HLS) jsou provedeny kroky *synthesis* (syntéza), *place-and-route* (umístění-a-pospojování) a *bitgen* (generace bitstreamu). [7]

Při použití HLS může vzniknout situace, že bude vytvořen algoritmus, který bude komplexní takovým způsobem, že jeho realizace na FPGA nebude možná. Oproti tomu při použití popisu pomocí structural/data flow circuits, je prakticky vždy algoritmus syntetizovatelný. [7]

Dalším negativním jevem přístupu HLS je situace, kdy dojde k vytvoření neoptimalizovaného komplexního algoritmu, který se ve vyšším programovacím jazyce jeví jako jednoduchý, ale při překladu do HDL a následných krocích *synthesis* -> *place-and-route* -> *bitgen* nebude možné vytvářený HW design do FPGA umístit, protože bude vyžadovat více *resources* (zdrojů LUTs, BRAM, atd.), než je v zařízení dostupných. [7]

V praxi je k tvorbě algoritmů stále více často využíváno vyšších programovacích jazyků a HLS, protože je tento přístup pro značný počet vývojářů SW srozumitelnější. Dalším častým přístupem v praxi je použití specializovaného SW jako je MATLAB™ a Simulink, který jsou schopen při použití odpovídajících balíčků přeložit vytvořený algoritmus do HDL, který je poté možné dále zpracovat a použít pro konfiguraci FPGA. Ovšem při využití přístupu se SW MATLAB™ je třeba mít k dispozici podporovaný HW, který obsahuje dostatečný počet zdrojů v FPGA struktuře. Tento přístup je značně finančně náročný v ohledu licence SW a taktéž vlivem vyšší ceny HW s větším počtem zdrojů.

4.3.2 Konverze HDL na konfigurační bitstream

V části *Forma tvorby algoritmu pro FPGA* byly představeny dvě hlavní formy tvorby algoritmu pro FPGA. Aby bylo možné algoritmy na FPGA „umístit“, je třeba vytvořenou rezprezentaci dále zpracovat.

Všechny vyšší úrovně reprezentace algoritmů jsou převedeny na HDL. Následným krokem je *syntéza* (*synthesis*), která slouží k převodu HDL na tzv. *netlist*. Při převodu je HDL převáděn na logické členy AND, OR apod. [7]

Po vytvoření netlistu je nutné rozhodnout, jakým způsobem je možné a výhodné realizovat jednotlivé bloky v logických buňkách a LUT. Konečné využití členů závisí na rozsahu vstupů realizovatelných LUT. Proces seskupování logických členů a určování funkce LUT se nazývá mapování (MAP). Výsledkem MAP je opět netlist. Tento netlist však reprezentuje FPGA členy (LUT, klopné obvody apod.). [7]

Po mapování následuje proces umístování (placement) při kterém je rozhodováno, které z logických bloků budou realizovat FPGA členy, získané v kroku MAP. [7]

Bloků, které jsou umístěny ve struktuře FPGA je nutné spojit pomocí dostupných propojů. Proces spojování a optimalizace propojů takovým způsobem, aby bylo minimalizováno časové zpoždění signálu,

se nazývá *routing*. Obvykle je proces slučován s MAP do jedné fáze a nazývá se *place-and-route* (PAR). [7]

Posledním krokem je vytvoření binárního souboru, nazývaného *bitstream*, kterým je poté konfigurováno FPGA. Tento proces převede netlist z kroku PAR na nastavení SRAM v jednotlivých logických buňkách FPGA tak, aby byl vytvořen požadovaný design v FPGA. Proces převede konfiguraci propojů a propojovacích matic do dalších SRAM, které ovládají příslušné propoje a matice. [7]



Obr. 4 - 4 Blokové schéma převodu aplikace naprogramované v procedurálním jazyce na bitstream, kterým je konfigurováno FPGA.

4.4 Spotřeba

FPGA je využíváno pro akceleraci aplikací pro svou nižší spotřebu energie než má CPU nebo GPUs. Ovšem oproti ASICs, má FPGA stále značnější spotřebu, proto stále probíhá výzkum, který má za cíl jejich energetickou náročnost snížit ale zachovat jejich výkon a spolehlivost.

Nižší potřebný výkon pro realizaci nepohonářské aplikace podporuje výzkum a článek [13], ve kterém autoři představují svoji práci, v níž realizovali hru. Ve hře je hlavním úkolem aplikace výpočet stínů a odrazů materiálů. Způsob vykreslení, který je v aplikaci použit, je nazýván *ray tracing*. Ray tracing je označován jako výpočetně náročný způsob, který není vhodný pro real-time aplikace ale pro vykreslování nepohyblivých obrazů, které není nutné zobrazovat v reálném čase. [14]

Autoři v textu popisují, že v případě využití FPGA pro výpočty v reálném čase byla jeho spotřeba 660 mW. Hru autoři vyzkoušeli spustit také na CPU platformě skládající se z Ryzen™ 4900H 8-core/16 threads 64-bit CPU @ up to 4,4 GHz clock. V případě testování na CPU byla indikována spotřeba 33 W. Tudíž při použití FPGA spotřeba klesla přibližně 50x. [13]

I přes nízkou spotřebu energie v FPGA jsou prováděny výzkumy, jak minimalizovat disipaci elektrické energie v podobě tepla a přiblížit se tak energetické náročnosti ASICs.

Disipace energie v FPGA je rozdělena na statickou a dynamickou. Statická disipace je způsobena zbytkovým proudem tranzistorů ve vypnutém stavu mezi drain a source elektrodou, mezi gate a drain elektrodou a jevem, nazvaným gate direct-tunneling. [15]

Dynamická disipace je způsobena spínacími a vypínacími ztráty použitých tranzistorů (obvykle CMOS) a je závislá na použitém napětí, frekvenci a kapacitě přechodů, kterou je třeba nabít a vybit při spínání a vypínání tranzistorů. [15]

4.5 Využití

Programovatelná logická hradlová pole se pro svoji nízkou spotřebu, vysoký výpočetní výkon a klesající cenu materiálů začínají využívat mnohem častěji v odvětvích, ve kterých bylo doposavad' využíváno CPU a GPUs. Aplikace FPGA je možné v rámci této práce rozdělit na nepohonářské a pohonářské.

4.5.1 Aplikace v nepohonářských odvětví

Díky univerzalitě FPGAs je možné jej využít v mnoha aplikacích v různých odvětvích. Stále se zvyšující požadavky na výpočetní výkon urychlují nasazování FPGAs do provozu, kde jsou v současné době

instalovány CPU nebo GPUs.

Poptávka po dostupnosti FPGA způsobila vznik Cloud služeb, které nabízí FPGA výkon on-demand. Jedním z významných poskytovatelů je Amazon Web Services (AWS), který nabízí FPGA akceleraci v Cloudu. Tuto službu ocení především aplikace, které nejsou vázány na reálný hardware ale pouze potřebují výpočetní výkon, který mohou v průběhu tvorby, debugingu či realizace aplikace měnit bez nutnosti pořizování výkonných a finančně náročných FPGA zařízení. Více informací o *Amazon EC2 F1 Instances* službě virtuálních FPGA je dostupné na [16].

Existuje mnoho výpočetně náročných aplikací jako jsou výpočty finančních modelů pro ekonomiku, výpočty simulací pro bioinformatiku, seismické modelování při hledání vzácných surovin apod., které je vhodné realizovat pomocí hardwarového akcelerátoru. Více informací o těchto výpočetně náročných aplikacích je možné získat v [17].

Na akceleraci zpracování audiovizuálních děl je převážně určeno GPUs. Ovšem pro aplikace, v nichž je vyžadováno zpracování obrazu v reálném čase s minimální spotřebou energie a nízkou hmotností zařízení, je často využíváno FPGA. Aplikace využití FPGA pro vozidla, která při své jízdě analyzuje okolní prostor jsou popsány v [18]. Tyto aplikace nesou souhrnný název „intelligent spaces applications“. Obvykle je pro analýzu okolního prostoru využíváno více kamer, z nichž každá obsahuje vlastní výpočetní jádro (FPGA). Díky tomu výpočetně náročné aplikace, jako např. analýza hloubky obrazu pro rozpoznání objektů, probíhá v FPGA a ostatní nenáročné výpočty a řízení v SW v CPU. [18]

Protože aktuálním trendem je snižování energetické náročnosti a zvyšování výpočetního výkonu, dochází neustále k vývoji nových aplikací, které využívají FPGA pro akceleraci výpočetně náročných kroků. Není možné všechny aplikace v tomto textu obsáhnout.

4.5.2 Aplikace v elektrických pohonech

V některých případech je elektrický pohon rozměrná a finančně náročná sestava, proto by zkoumání určitých kritických stavů těchto soustav mohlo být ekonomicky i technicky nevýhodné. V tomto případě je vhodné vytvořit přesný matematický model jednotlivých analyzovaných součástí a náročné výpočty modelu akcelerovat pomocí FPGA. Na základě odezvy modelu je poté možné analyzovat stavy, které by v případě realizace na reálném stroji mohly způsobit jeho destrukci či částečnou ztrátu funkčnosti. Proto se v průmyslu využívá Hardware-in-the-loop simulation (HIL), kdy je vytvořen matematický model, který poskytuje elektrické signály do testovaného systému a na základě jeho reakce je možné vyhodnotit, jakým způsobem by se choval reálný modelovaný systém. [18] [19]

Kromě FPGA simulace je možné FPGA využít také pro řízení elektrických pohonů. Možnosti realizace řízení AC elektrických strojů pomocí FPGA a analogově digitálních převodníků (ADC) jsou prezentovány v [20]. V dokumentu jsou popisovány tři realizace řízení, resp. regulace pohonu. Regulace realizována pomocí hystérézních on-off regulátorů, PI regulátorů a prediktivních regulátorů. [20]

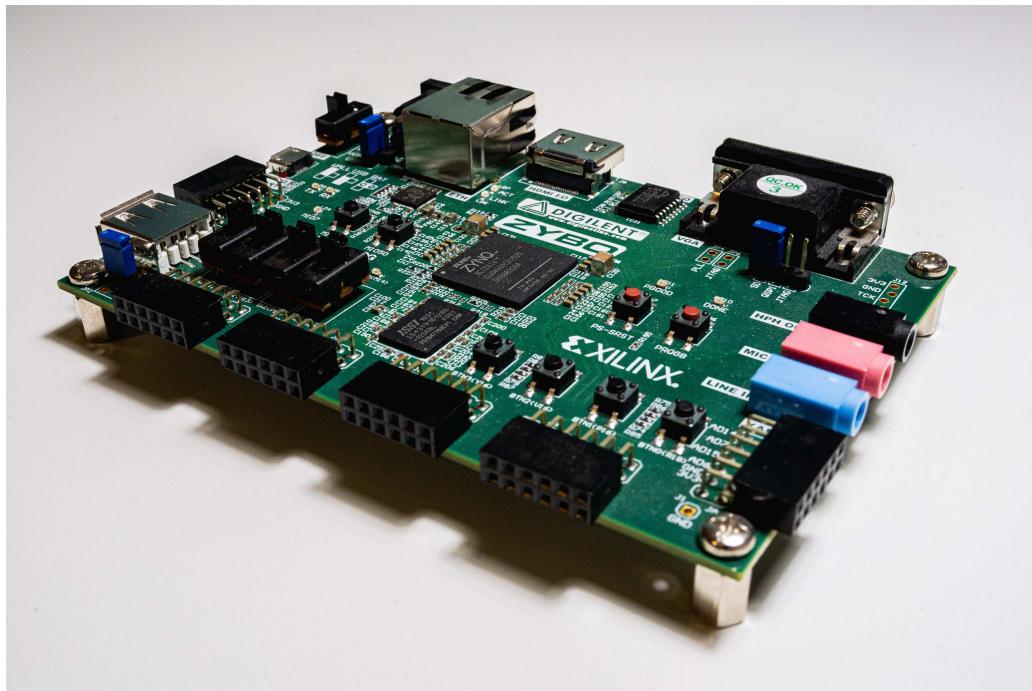
Všechny prezentované způsoby regulace v [20] byly před syntézou realizovány v prostředí MATLAB™ a Simulink. Tento způsob tvorby modelů a algoritmů je v praxi upřednostňován, protože umožňuje expertům na řízení a regulaci bez znalostí mikroelektroniky, programování v HDL a způsobu fungování FPGA pracovat na dané problematice. Oproti tomu je třeba zvážit, jaké jsou požadavky na rychlosť, výkonnost a optimalizované řízení aplikace a zdali použití předpřipravených knihoven a zjednodušených nástrojů nebude mít příliš značný vliv na rychlosť výpočtu a tudíž zpracování dat a řízení v reálném čase. [20]

5 Vývojová deska Digilent Zynq

Vývoj akcelerovaných aplikací je možné realizovat na relativně velikém množství dostupného HW. V některých případech je design vývojových desek dokonce výrobcem uveřejňován a tudíž v případě dostatečných znalostí je možné si sestavit vlastní HW z dostupných komponent takovým způsobem, aby vyhovoval požadované embedded aplikaci. Výhodné ovšem je využít již připravená řešení vývojových desek, které zjednoduší tvorbu a debugging aplikace.

Při tvorbě této práce byl realizován první vývoj a seznámení s prostředím akcelerovaných aplikací na vývojové desce *Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board* od firmy Digilent. [21] Jedná se o model vývojové desky, který byl na trhu nahrazen novějšími variantami s označením *ZYBO Z7-10* a *ZYBO Z7-20*, které jsou stále v aktivním prodeji. Hlavním rozdílem desek je verze ZynQ čipu, který v moderních deskách disponuje ARM procesorem s vyšší taktovací frekvencí a s modernějším FPGA s vyšším počtem LUTs, klopných obvodů a s rozsáhlější pamětí RAM. Bližší porovnání specifikací těchto desek je dostupné na [22].

V další části textu jsou představeny významné komponenty vývojové desky *Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board*.



Obr. 5 - 1 Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – boční pohled.

5.1 Základní přehled

5.1.1 CPU a FPGA čip

Hlavní částí vývojové desky je čip, obsahující FPGA a CPU jednotky zakomponované v jedné polovodičové struktuře. Jak již bylo zmíněno v části *Hardware Accelerated Applications*, tato struktura se nazývá heterogenní.

Deska obsahuje čip Xilinx Zynq-7000 (typ XC72010), který umožňuje pro vývoj aplikací použít SDK od firmy Xilinx. V tomto čipu je integrován dvou jádrový procesor ARM Cortex-9, který slouží pro řízení akcelerovaných aplikací na Xilinx FPGA sedmé série. Detailní schéma blokové architektury SoC

s označením sběrnic a komunikace jednotlivých částí čipu je zobrazeno na obr. 5 - 2.

Z naznačené architektury je možné vyvodit, že se SoC skládá ze dvou hlavních částí, které je možné dále rozdělit na jednotlivé bloky:

- Processing System (PS),
 - Application processor unit (APU),
 - Memory interfaces,
 - I/O peripherals (IOP),
 - Interconnect,
- Programmable Logic (PL).

Blok PS

Blok PS se skládá z dílčích bloků, které neslouží k akceleraci aplikací, ale k běhu host programu (program běžící na PS). Blok PS reprezentuje prakticky celou architekturu čipu vyjma části věnované PL.

Blok APU

Blok APU obsahuje CPU Cortex-A9 a další podpůrné bloky jako např. přímý přístup do paměti (DMA controller), General interrupt controller (GIC) pro maskování a ovládání přerušení, watchdog a další podpůrné bloky.

Blok Memory interfaces

Memory interfaces slouží k přístupu APU a PL k pamětím typu DDR3, DDR3L, DDR2 a LPDDR-2. Je možné také vybrat, zda šířka sběrnice bude 16, nebo 32 bitů. K dispozici jsou zakomponované kotroléry přenosu dat pro optimalizaci rychlosti, Static Memory Controller nebo Quad-SPI Controller.

Blok IOP

IOP se skládá ze standardizovaných rozhraní vhodných pro průmyslovou komunikaci. Obsahuje např. GPIO, Gigabit Ethernet, dva bloky USB Controller, dva bloky SD/SDIO Controller pro bootování SD karty, dva bloky SPI Controller, dva bloky CAN Controller, dva bloky UART Controller a dva bloky I2C Controller.

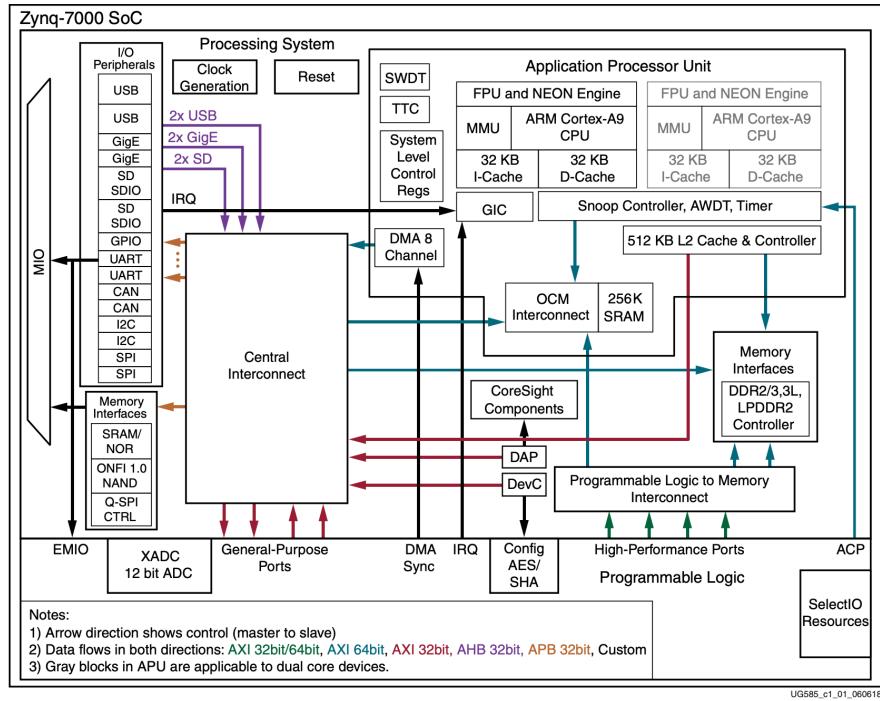
Blok Interconnect

Blok Interconnect, resp. na obr. 5 - 2 označený Central Interconnect slouží k propojení jednotlivých bloků SoC dle požadované technologie a rychlosti.

Blok PL

Blok PL reprezentuje logické programovatelné pole (FPGA), v němž jsou zakomponovány další podpůrné prvky jako např. blok zpracování digitálních signálů, řízení taktovacích hodin, analogově digitální převodník (ADC).

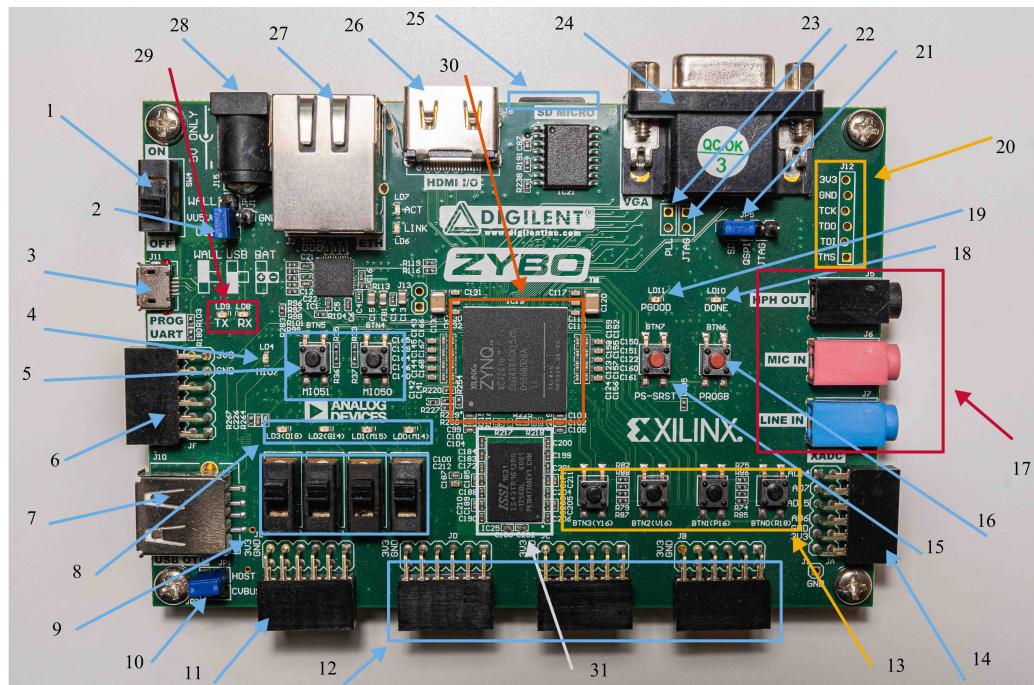
Detailní technické specifikace, složení a parametry jmenovaných bloků jsou uvedeny v [23].



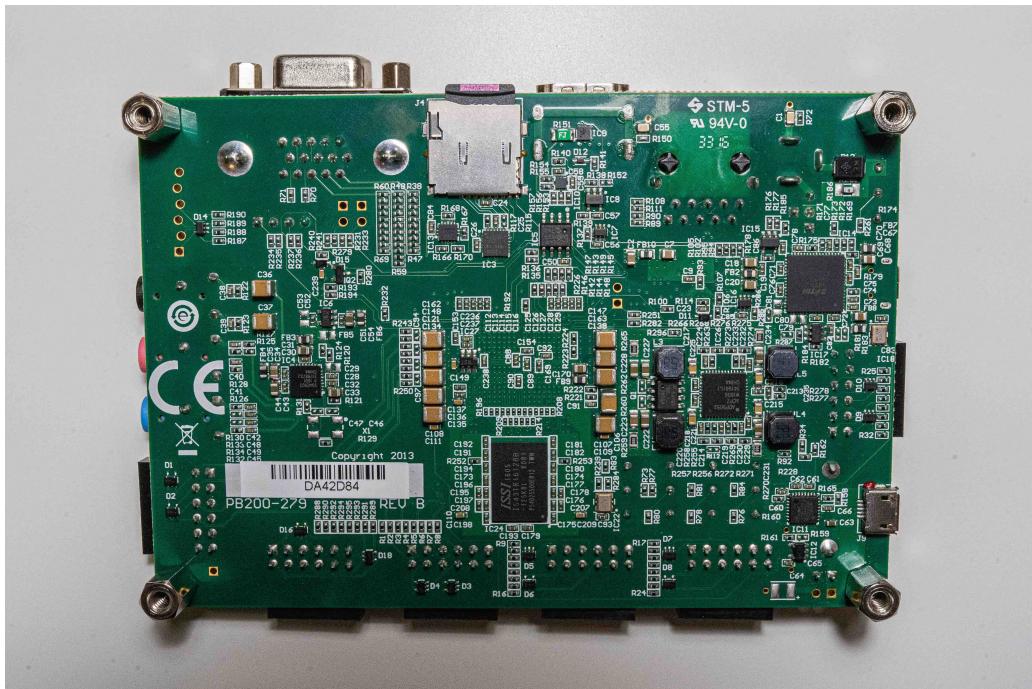
Obr. 5 - 2 Detailní schéma čipu Zynq-7000, umístěného na vývojové desce Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board. (převzato z [23])

5.1.2 Uspořádání vývojové desky Zybo Zynq-7000

Na obr. 5 - 3 je zobrazen horní pohled na vývojovou desku, na kterém jsou vyznačeny významné části, jimž je vhodné věnovat pozornost. Číselné označení koresponduje s označením a vysvětlivkou v tabulce 5 - 1. Pro úplnost je spodní strana desky zobrazena na obr. 5 - 4.



Obr. 5 - 3 Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – vrchní pohled s vyznačením komponent.



Obr. 5 - 4 Vývojová deska Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board – spodní pohled.

Tab. 5 - I Popis označených komponent na vývojové desce Digilent Zybo Zynq-7000. (informace a značení převzaty z [24])

označení	popis	poznámka
1	Power Switch	galvanické sepnutí napájecího obvodu
2	Power Select Jumper and Battery Header	výběr napájecího vstupu konektor, USB, baterie
3	Shared UART/JTAG USB port	komunikace UART a JTAG debugging
4	MIO LED	multiplexed LED – možnost výběru signálu
5	MIO Pushbuttons (2)	multiplexed input
6	MIO Pmod	možnost připojení periférií
7	USB OTG Connectors	USB port typ A/micro USB (spodní část)
8	Logic LEDs (4)	zobrazování 1/0
9	Logic Slide Switches (4)	logický vstup 1/0
10	USB OTG Host/Device Select Jumpers	výběr módů zařízení
11	Standard PMOD	chráněné PMOD, limitace max. přenosu informace
12	High-speed PMODs (3)	jako standard ale bez ochrany, vyšší rychlosť
13	Logic Pushbuttons (4)	logický vstup 1/0
14	XADC PMOD	možnost analog/digi input/output, spojeno s ADC v Zynq
15	Processor Reset Pushbutton	reset PL, paměti v PS
16	Logic Configuration reset Pushbutton	reset PL, zrušení DONE informace
17	Audio Codec Connectors	stereo line in, mono mikrofon, stereo output
18	Logic Configuration Done LED	signál o úspěšném dokončení konfigurace PL
19	Board Power Good LED	1/0, 1 – nominální napětí na všech sběrnících
20	JTAG Port for optional external cable	externí JTAG
21	Programming Mode Jumper	výběr „programovacího vstupu“, SD karta, QSPI, JTAG
22	Independent JTAG Mode Enable Jumper	JTAG mimo PS, viditelné pouze PL
23	PLL Bypass Jumper	přemostění PLL (CLK), pro možnost konfigurace PLL
24	VGA connector	připojení displeje
25	microSD connector	na spodní straně
26	HDMI Sink/Source Connector	input/output, nutné implementovat encoding a decoding v logice
27	Ethernet RJ45 Connector	komunikace
28	Power Jack	napájení 5 V/2,5 A
29	TX/RX LED	indikace UART komunikace
30	Xilinx Zynq SoC	systém na čipu
31	DDR2 Memory	RAM

6 Vývojová deska Xilinx Kria KR260

Deska od firmy Digilent, představená v části *Vývojová deska Digilent Zybo*, je vhodná pouze pro prvotní seznámení s procesem vytváření akcelerovaných aplikací. Pro náročnější aplikace, které vyžadují využití většího množství LUTs, nebylo v této práci možné Zybo použít. Naopak PL vývojové desky Kria KR260 obsahuje dostatečné množství LUTs a díky svým moderním komponentám, může být efektivně využita k tvorbě náročnějších aplikací.

Hlavní částí vývojové desky KR260 je „modul“ *Kria K26 System-on-Module*. Tudíž oproti Digilent Zybo, které využívá SoC, deska KR260 využívá SOM. Přednosti jednotlivých architektur byly již představeny v části *System on a chip* a *System on modules*. Po ukončení vývoje aplikace na vývojové desce (a popřípadě po ukončení vytváření návrhu CC) je možné pro aplikaci v průmyslu zakoupit samostatný modul na BB ve vhodné variantě, a umístit ho na danou CC. V této práci je využíván standardní vývojový „Starter kit“ s deskou KR260, jejíž komponenty je vhodné představit.



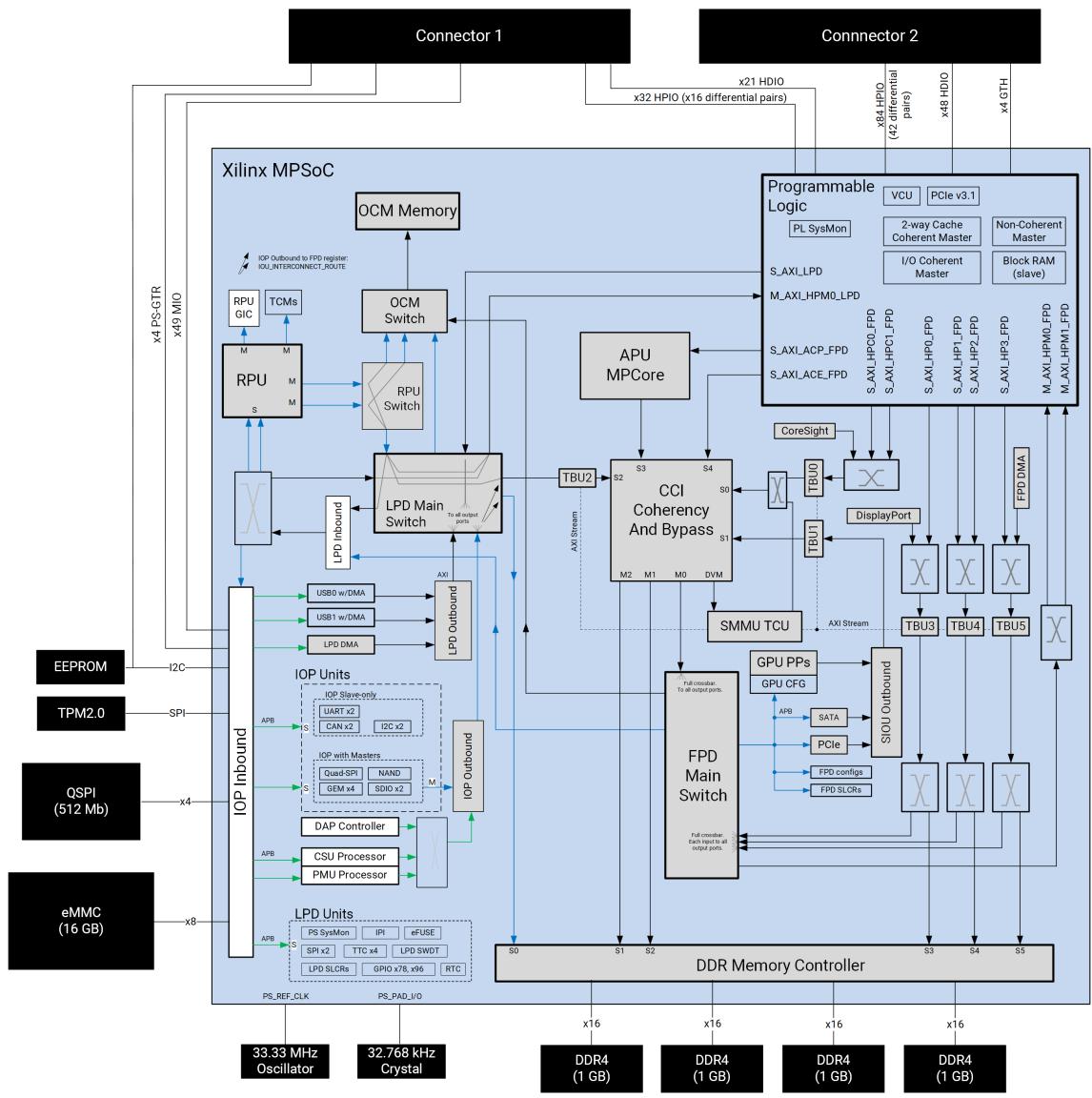
Obr. 6 - 1 Vývojová deska Xilinx Kria KR260 – boční pohled.

6.1 Základní přehled

6.1.1 CPU a FPGA čip

Strukturu SOM je možné popsat jako modernější a komplexnější. Základní struktura obsahuje bloky PS a PL. Ukázkový blokový diagram struktury udávané výrobcem je na obr. 6 - 2.

Největšími rozdíly mezi použitými SoC Zybo a SOM Kria je např. velikost operační paměti, počet jader, taktovací frekvence procesorů v PS, počet LUTs nebo logických buněk v FPGA (PL). Úplné specifikace pro K26 SOM je možné nalézt v [5].



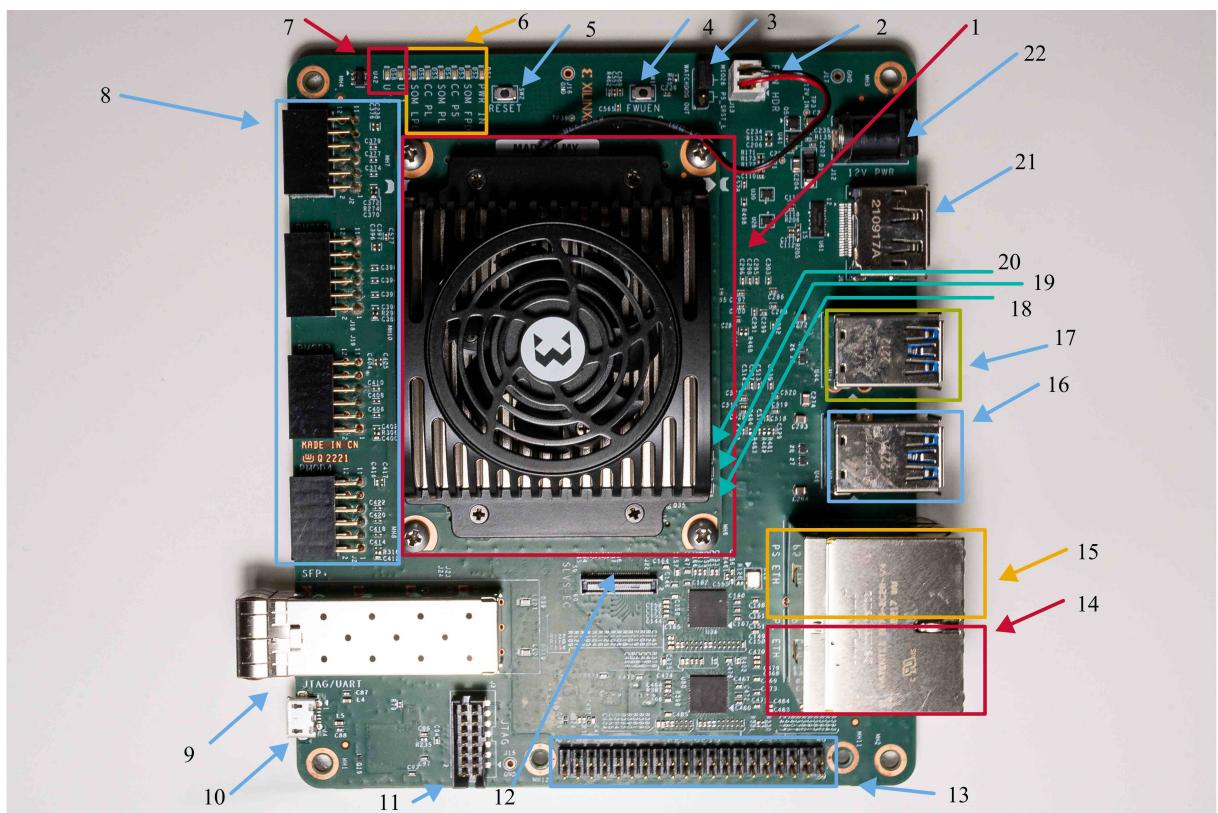
Obr. 6 - 2 Blokový diagram K26 SOM Kria. [5]

6.2 Uspořádání vývojové desky

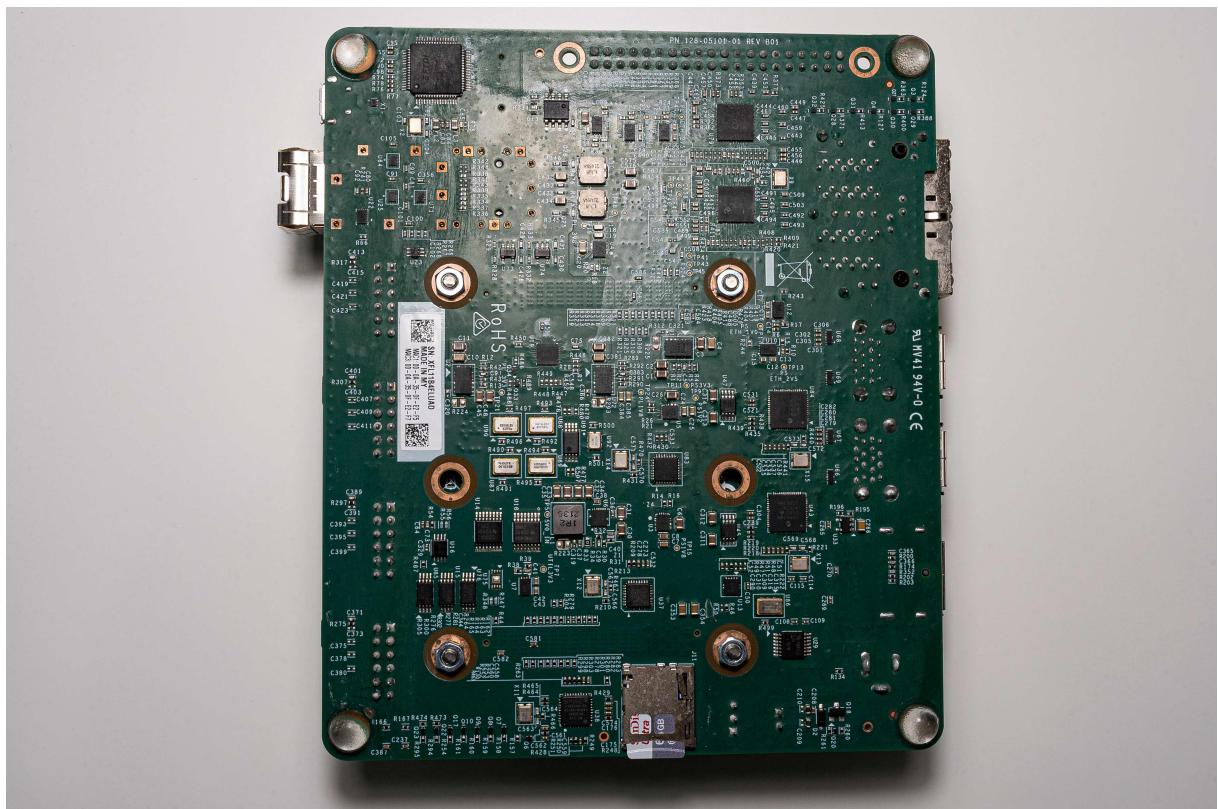
Na obr. 6 - 3 je zobrazen horní pohled na vývojovou desku, na kterém jsou vyznačeny významné části, jimž je vhodné věnovat pozornost. Číselné označení koresponduje s označením a vysvětlivkou v tabulce 6 - 1. Na spodní straně desky je umístěn slot pro SD kartu, na kterou je umisťován (flashován) operační systém pro PS. Pro úplnost je spodní strana desky zobrazena na obr. 6 - 4.

Tab. 6 - 1 Popis označených komponent na vývojové desce Xilinx Kria KR260. (informace a značení převzaty z [25])

označení	popis	poznámka
1	SOM modul na BB a Fansink	-
2	J13 Fan Power	napájení větráku chladiče
3	J1 watchdog	-
4	SW1 Firmware Update	-
5	SW2 Reset	-
6	DS1–DS6 Power Status LEDs	pokud vše ok, zbarveny zeleně
7	DS7–DS8 (UF1 a UF2) Uživatelsky ovládané LED	-
8	J2, J18, J19, J20 PMOD konektory	-
9	J23 a J24 SFP+	optický konektor
10	J4 Micro USB	UART/JTAG
11	J3 PC4 JTAG	-
12	J22 SLVS-EC	konektor pro připojení kamery
13	DS36 Raspberry Pi HAT	-
14	J10A, J10B RJ-45 PL Ethernet	konektor připojen přes PS
15	J10C, J10D RJ-45 PS Ethernet	konektor připojen do PS
16	U46 USB3.0	-
17	U44 USB3.0	-
18	DS36 PS Status LED	-
19	DS35 HeartBeat LED	-
20	DS34 PS Done LED	-
21	J6 DisplayPort	-
22	J12 DC Jack	-



Obr. 6 - 3 Vývojová deska Xilinx Kria KR260 vrchní pohled s vyznačením komponent.



Obr. 6 - 4 Vývojová deska Xilinx Kria KR260 – spodní pohled.

6.2.1 Dostupné K26 SOM

Moduly Kria K26 SOM jsou dostupné v několika variantách. V roce 2022 jsou dostupné varianty *Commercial* a *Industrial*. Další dělení dvou hlavních variant je SOM s povoleným nebo zakázaným šifrováním. Varianty se odlišují označením Encryption Disabled (ED) a Encryption Enabled (-). Pokud je šifrování povoleno, je možné šifrovat konfigurační soubory a nebo ve vytvářených aplikacích využívat implementované *crypto-accelerator* bloky. Encryption Enabled varianty jsou v některých zemích zakázané a proto je důležité při výběru zboží dbát pokynů prodejce.

Významné rozdíly *Commercial* a *Industrial* variant jsou uvedeny v tab. 6 - 2.

Tab. 6 - 2 Porovnání hlavních parametrů Kria K26 SOM Commercial a Industrial. (informace a značení převzaty z [26])

parametr	K26 Commercial SOM	K26 Industrial SOM
pracovní teplota	0–85 °C	-40–100 °C
záruka	2 roky	3 roky
předpokládaná doba životnosti produktu	5 let	10 let
dostupnost produktu	10 let	10 let

Vývojová deska Xilinx Kria KR260 obsahuje dle informací výrobce SOM, jež není určen pro nasazení do produkčních aplikací (non-production) a je v teplotní třídě *Commercial*. [5]

7 Porovnání představených SoC/SOM platforem pro řízení elektrických pohonů

V předchozích kapitolách byly představeny dvě smysluplné (z finančního i aplikačního hlediska), komerčně dostupné platformy, které je možné využít pro vývoj aplikací. Výběr byl zaměřen na SoC a SOM, umístěných na vývojových deskách, které je možné využít pro vývoj požadované aplikace. Často po prvním vývoji aplikace následuje uvědomění, jaké periferie by měla CC obsahovat. Poté je možné pro velkoprodukci dané aplikace začít vyvíjet vlastní CC a řešit integraci čipu/modulu na Printed Circuit Board (PCB). V této práci byly však využity již připravené vývojové desky, které je možné vzájemně porovnat v několika ohledech.

7.1 Konektivita

Aby bylo možné komunikovat s řízeným zařízením, tudíž vysílat řídící signály a získávat informace o jeho stavu, je při hodnocení důležitým faktorem možnost připojení.

Obě představené platformy disponují minimálně čtyřmi PMOD konektory s 12 piny (2x U+, 2x GND, 8x I/O), pomocí kterých je možné připojit senzory, převodníky nebo naopak ovládat drivery spínacích polovodičových součástek či výkonových polovodičových můstků. PMOD konektory využívá mnoho komerčně dostupných prvků jako jsou senzory, H můstky nebo ADC/DAC převodníky.

Dalším důležitým faktorem konektivity je připojení pomocí Ethernetu. Xilinx Kria KR260 obsahuje 4 konektory, přičemž dva jsou připojeny přímo do PS a dva do PL. Digilent Zybo obsahuje pouze jeden Ethernet konektor.

Pro připojení periferií nebo externích datových úložišť je možné využít konektor USB. Protože Digilent Zybo Z7 je staršího data vydání, využívá pouze USB 2.0, zatímco Xilinx Kria KR260 využívá připojení pomocí USB 3.0.

Pro připojení externího displeje je možné u Zybo využít D-SUB (VGA) konektor. U Kria KR260 novější Display Port.

Digilent Zybo má na svém PCB umístěny konektory pro výstup reproduktorů či vstup mikrofonu. Deska KR260 tyto konektory neobsahuje.

Značným přínosem pro konektivitu Kria KR260 je Raspberry Pi hardware attached on top (HATs) konektor, jež umožňuje připojení rozšiřovacích desek, určených pro Raspberry Pi.

Pro AI aplikace na KR260, které využívají externího obrazu, je možné připojit externí webkameru pomocí konektoru SLVS-EC.

Posledním významným konektorem na CC Xilinx Kria je SFP konektor pro připojení optických vláken.

7.2 PS a PL

Protože je porovnávaná deska Digilent Zybo a Xilinx Kria KR260 různého data vydání a také na jejich pořízení je třeba rozdílných finančních prostředků, odpovídají zdroje pro PS a PL daným cenám.

Jak již bylo zmíněno v sekci *Vývojová deska Digilent Zybo*, PS vývojové desky Digilent Zybo Zynq-7000 obsahuje dvoujádrový procesor Cortex-A9 s taktovací frekvencí 650 MHz. Oproti tomu novější PS desky s Kria K26 SOM obsahuje čtyřjádrový procesor Cortex®-A53 MPCore™ s taktovací frekvencí až 1,5 GHz. SOM je doplněn dvoujádrovým real-time procesorem Arm Cortex-R5F MPCore s taktovací frekvencí až 600 MHz. [24] [5]

Dalším důležitým faktorem jsou zdroje pro PL. Digilent Zybo disponuje pouze 17 600 LUTs. Oproti

tomu K26 SOM obsahuje 117 120 LUTs. Novější verze Digilent Zybo desek obsahuje novější verze Zynq čipu, který nabízí také 17 600 LUTs (Zybo Z7-10) nebo 53 200 LUTs (Zybo Z7-20). [24] [5]

Počet LUTs v této práci má značný vliv na možný rozsah vytvářené aplikace, která je počtem zdrojů (LUTs, Flip-Flops, BRAM atd.) velmi ovlivněna.

Vlivem omezených zdrojů bylo možné i přes optimalizaci C++ kódu (pro dodržení load-compute-store modelu programování [27]) ukázkové akcelerované aplikace (kernelu) v této práci umístit do PL v Digilent Zybo Z7 pouze I - n model asynchronního motoru, jehož výsledkem byl transformační úhel Θ a složky vektoru magnetického toku rotoru $\underline{\psi}_2^{\alpha\beta}$. Oproti tomu při využití Kria K26 SOM je možné využít PL pro výpočet I - n modelu stroje, zjednodušeného modelu asynchronního motoru, regulátorů a invertoru.

7.3 Developer Experience

V moderní době, kdy je kladen značný důraz na rychlosť vývoje aplikace, je hodnotícím faktorem i developer experience (DX). Tudíž jak je systém konfigurace a vytváření aplikací přívětivý pro vývojáře. U Digilent Zybo Z7 byl používán postup vytvoření operačního systému *PetaLinux* již s pevně daným Device Tree (DT), které bylo možné měnit pomocí rekonfigurace a následného opakování celého procesu tvorby systému a aplikace. To přinášelo značné časové prodlevy při ladění aplikace a vytvářeního PL hardware.

Při použití Xilinx Kria K26 SOM je možné při tvorbě systému definovat kostru DT pro PS, kterou je poté možné do značného rozsahu upravovat pomocí Device Tree Overlay (DTO). Pomocí DTO je možné rekonfigurovat IP vytvářené v PL. Změnou v DTO je možné ovlivňovat funkčnost některých IP v PL při chodu operačního systému *PetaLinux*. Ovšem tyto úpravy mají určitá omezení a je vhodné Device Tree vhodně nakonfigurovat již při vytváření operačního systému *PetaLinux*.

Více informací o chování a tvorbě DT.DTO, zjištěných při realizaci této práce, je uvedeno v části *Konfigurace Device Tree*.

Digilent pro své výrobky vytvořil „board files“ a „constraints files“, které umožňují snazší konfiguraci PS a PL v prostředí Vivado. Značným přínosem jsou „constraints files“, které umožňují snazší a rychlejší mapování fyzických pinů vývojové desky k portům, pinům a rozhraní, vytvářených ve Vivado. Pro vývojovou desku Xilinx Kria KR260 jsou v repozitáři ve Vivado již „board files“ obsaženy. Ovšem oficiální „constraints files“ nejsou od výrobce k dispozici. Pro mapování pinů je nutné si vyžádat dokumentaci, pomocí které je možné odvodit požadované mapování a „constraints files“ vytvořit. Potřebná dokumentace pro odvození mapování je v souborech [28] a [29].

7.4 Aplikace a operační systém

Aplikace pro Digilent Zybo je možné vytvářet jako *Bare Metal / Standalone* nebo jako aplikace pro operační systém *PetaLinux*.

Pro Xilinx Kria je možné využít *Bare Metal / Standalone* přístup, *PetaLinux* a také distribuci operačního systému Linux *Ubuntu*. Výrobce na stránkách Wiki podpory produktů Xilinx zmiňuje, že poskytuje podporu převážně ve formě veřejného fóra na adrese support.xilinx.com. Oficiální podpora je vždy dostupná pro dvě poslední major verze softwarových nástrojů, které jsou součástí *PetaLinux Tools* a *Xilinx SDK*. [30]

Výrobce vytvořil pro Xilinx Kria ukázkové akcelerované aplikace, které je možné při využívání operačního systému *Ubuntu* přímo stáhnout z Kria App Store. V plánu výrobce je vytvořit více aplikací, které by mohly sloužit jako výchozí bod pro vývojáře a byly by umístěny v Kria App Store. [31]

PS na porovnávaných vývojových deskách podporují PREEMPT_RT Linux Patch. K tomuto patchi ovšem Xilinx, Inc. neposkytuje žádnou oficiální podporu a je nutné získávat informace přímo od autorů projektu na Wiki stránce [32] nebo z omezené podpory pomocí fóra. Více informací o PREEMPT_RT Linux Patch *PetaLinux* je v sekci *RealTime Linux Patch*.

8 Zpětnovazební vektorová regulace

V této práci bude využití platformy demonstrováno pomocí realizace simulace zpětnovazebního vektorového řízení asynchronního motoru. Zadání úlohy bylo převzato z předmětu B1M14EPT. [33]

Hlavním záměrem této práce není představovat princip vektorové regulace, ale je vhodné pomocí blokového schématu nastínit její princip. Na obr. 8 - 1 je zobrazeno obecné schéma zpětnovazební vektorové regulace. Naznačeným obecným způsobem by bylo možné realizovat fyzickou regulaci s použitím vývojové desky a potřebných prvků (ADC, senzorů, ...).

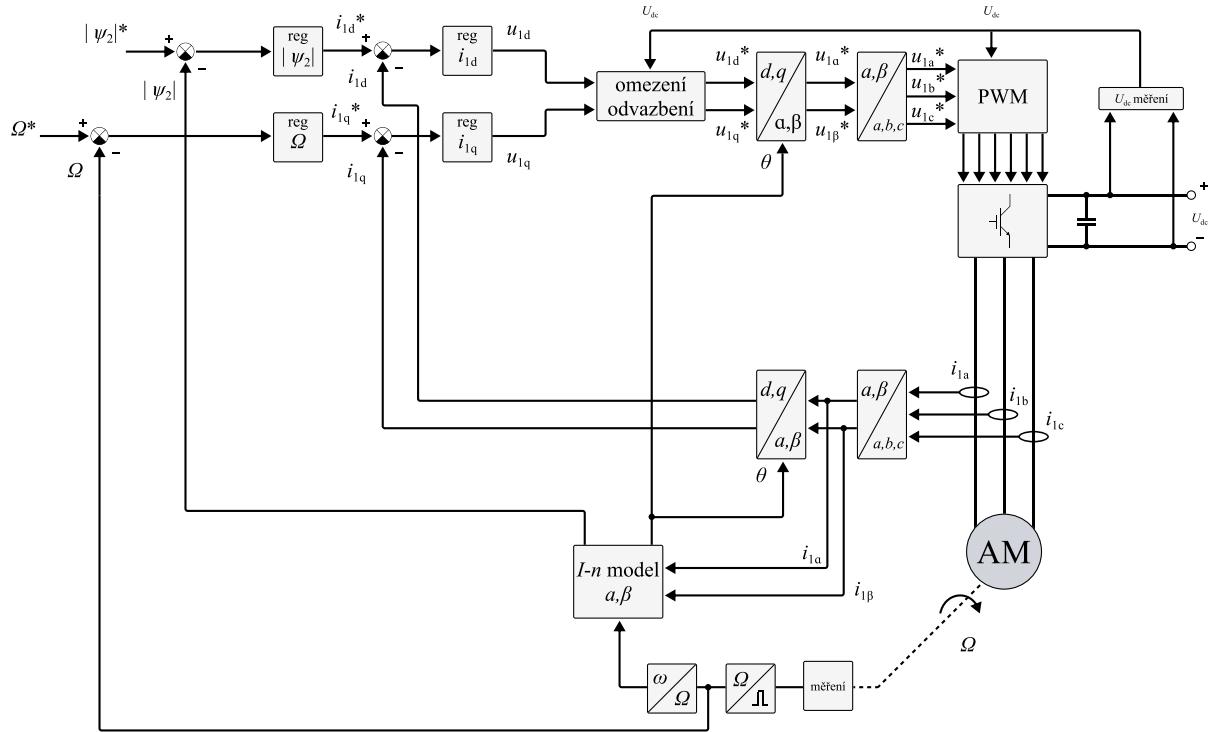
Z finančních důvodů je možné při spojení vinutí motoru do hvězdy a volby konstanty Clarkovi transformace $K = 2/3$ měřit pouze dvě hodnoty napájecích proudů statoru motoru. Pro úplnost jsou naznačeny v obr. 8 - 1 všechny tři snímače. Pro měření otáčivé rychlosti motoru je možné použít inkrementální snímač a jeho výsledné impulzy zpracovávat pomocí PS nebo PL. Pokud by byl k dispozici jiný druh snímače, je preferován prvek s možností komunikace pomocí SPI, která splňuje požadavky na rychlé příjemání a odesílání hodnot.

V této práci byla realizována pouze simulace jednotlivých prvků FOC. Upravené blokové schéma simulace je zobrazeno na obr. 8 - 2.

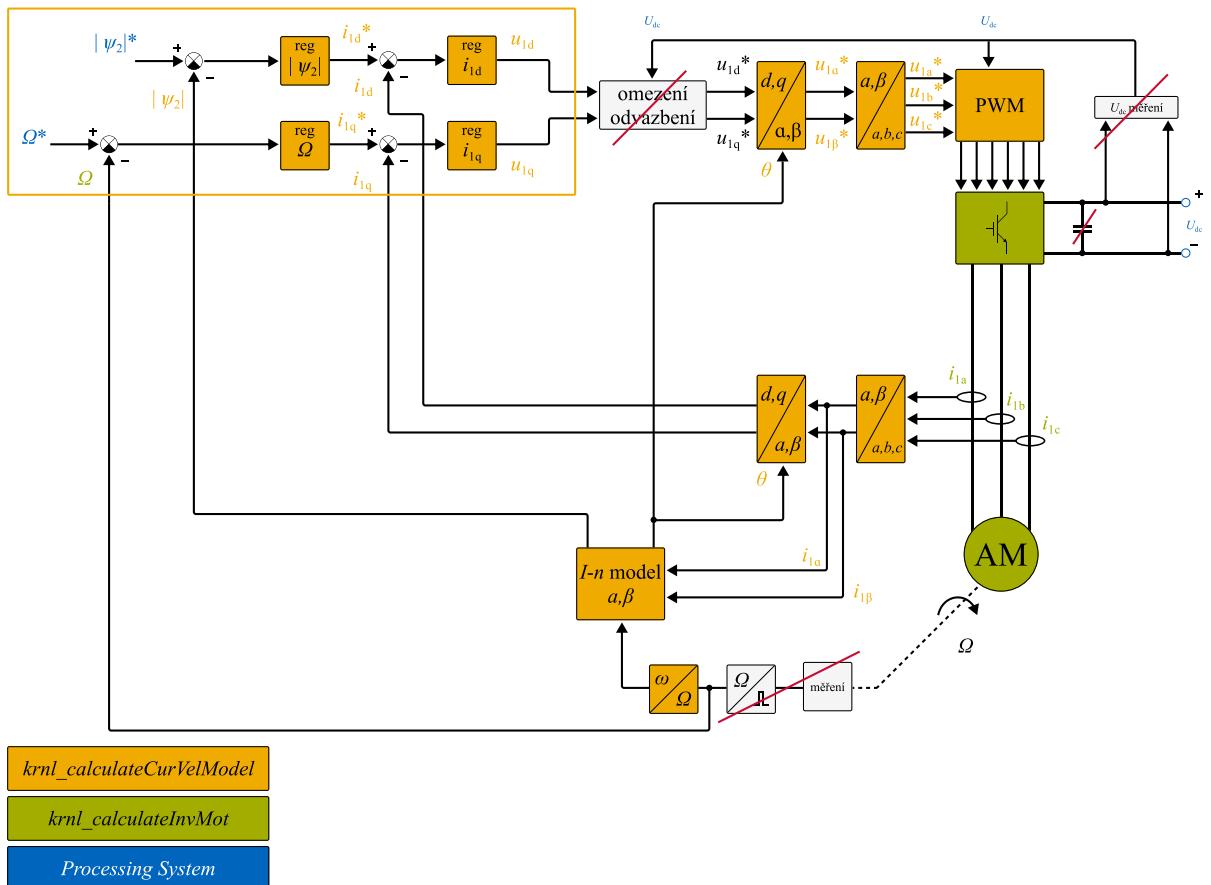
Žluté označení symbolizuje, že se výpočty provádějí v `krnl_calculateCurVelModel`, tudíž v prvním ze spouštěných kernelů v podaplikaci *CPU/FPGA Model*.

Zelené označení symbolizuje provádění výpočtů výstupního napětí invertoru a modelu asynchronního motoru v kernelu `krnl_calculateInvMot`. **Modré označení** blok symbolizuje, že je algoritmus prováděn v PS. **Modré označená** proměnná je nastavována v PS a **žlutě označená**, resp. **zeleně označená** je výstupem kernelu s *I-n* modelem, resp. s modelem asynchronního motoru.

Blok (omezení a odvazbení) a měření napětí U_{dc} nebyly v aplikaci *CPU/FPGA Model* implementovány. Bloky pro zpracování rychlosti otáčení byly vyneschány z důvodu, že mechanická otáčivá rychlosť je přímo jeden z výstupů modelu asynchronního motoru.



Obr. 8 - 1 Obecné schéma zpětnovazební vektorové regulace. (převzato z [33], [34], upraveno)



Obr. 8 - 2 Simulační schéma zpětnovazební vektorové regulace, realizované v této práci. (převzato z [33], [34], upraveno)

9 Model stroje

Jak již bylo představeno v předchozích částech textu, akcelerované aplikace v FPGA je možné použít na různé účely. Součástí této práce je realizace akcelerovaného výpočtu matematického modelu stroje. V této práci bude k demonstraci funkčnosti využit matematický model asynchronního motoru. Asynchronní motor bude modelován pomocí zjednodušeného modelu, zanedbávající některé jevy a pro FOC pomocí I - n modelu.

V části *PS a PL* je uvedeno, že vlivem omezených zdrojů je možné realizovat v Digilent Zybo Z7 pouze I - n model stroje. Ostatní výpočty je nutné realizovat v PS. V Xilinx Kria K26 je díky většímu množství zdrojů možné realizovat větší část modelu v PL a pomocí PS řešit pouze konfiguraci, řízení procesu akvizice dat nebo komunikaci.

9.1 Matematický popis „kompletního“ modelu stroje

Parametry motoru, využívaného v simulaci, byly získány z výukových materiálů předmětu B1M14EPT [33] a jsou uvedeny v tab. 9 - 1 a v tab. 9 - 2.

Motor je umístěn v laboratoři č. H-26 na Fakultě elektrotechnické Českého vysokého učení technického v Praze.

Tab. 9 - 2 Změřené parametry stroje.

Tab. 9 - 1 Štítkové údaje stroje.

P_n	12 kW
U_n	380 V
I_n	22 A
n_n	1460 min^{-1}
f_n	50 Hz
$\cos(\varphi_n)$	0.8
p_p	2

R_1	370 mΩ
R_2	225 mΩ
$L_{1\sigma}$	2,27 mH
$L_{2\sigma}$	2,27 mH
L_m	82,5 mH
L_1	84,77 mH
L_2	84,77 mH
J	0,4 kg·m ²

Kde P_n (W) je jmenovitý výkon stroje, I_n (A) je jmenovitý fázový proud stroje (efektivní hodnota), U_n (V) je jmenovité sdružené napájecí napětí stroje (efektivní hodnota), f_n (Hz) je jmenovitá napájecí frekvence stroje, $\cos(\varphi_n)$ (-) je jmenovitý účinník stroje, n_n (min^{-1}) jsou jmenovité otáčky stroje, p_p (-) je počet polpárů stroje, R_1 (Ω), resp. R_2 (Ω) je rezistivita statorového, resp. rotorového vinutí, $L_{1\sigma}$ (H), resp. $L_{2\sigma}$ (H) je statorová, resp. rotorová rozptylová indukčnost stroje, L_m (H) je magnetizační indukčnost stroje, L_1 (H), resp. L_2 (H) je statorová, resp. rotorová indukčnost, J ($\text{kg} \cdot \text{m}^2$) je moment setrvačnosti hřídele.

Použitý model je založen na výpočtu složek vektorů statorového proudu \underline{i}_1^k a rotorového toku $\underline{\psi}_2^k$ v souřadnicovém systému $\alpha\beta$ spojeném se statorem. Tudíž při použití $\omega_k = 0$. Bude volena konstanta $K = 2/3$. Poté bude stavový popis systému vypadat následovně. [33]

$$\frac{d}{dt} \begin{bmatrix} i_{1\alpha} \\ i_{1\beta} \\ \psi_{2\alpha} \\ \psi_{2\beta} \end{bmatrix} = \begin{bmatrix} -\frac{R_2 L_m^2 + L_2^2 R_1}{\sigma L_1 L_2^2} & 0 & \frac{L_m R_2}{\sigma L_1 L_2^2} & \frac{L_m}{\sigma L_1 L_2} \omega \\ 0 & -\frac{R_2 L_m^2 + L_2^2 R_1}{\sigma L_1 L_2^2} & -\frac{L_m}{\sigma L_1 L_2} \omega & \frac{L_m R_2}{\sigma L_1 L_2^2} \\ \frac{L_m R_2}{L_2} & 0 & -\frac{R_2}{L_2} & -\omega \\ 0 & \frac{L_m R_2}{L_2} & \omega & -\frac{R_2}{L_2} \end{bmatrix} \begin{bmatrix} i_{1\alpha} \\ i_{1\beta} \\ \psi_{2\alpha} \\ \psi_{2\beta} \end{bmatrix} + \begin{bmatrix} \frac{1}{\sigma L_1} & 0 \\ 0 & \frac{1}{\sigma L_1} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_{1\alpha} \\ u_{1\beta} \end{bmatrix}. \quad (9 - 1)$$

Stavový popis je vhodné doplnit o další rovnice, jež budou v simulaci využity.

$$M = \frac{3}{2} p_p \frac{L_m}{L_2} (\psi_{2\alpha} i_{1\beta} - \psi_{2\beta} i_{1\alpha}), \quad (9 - 2)$$

$$M - M_z = J \frac{d\Omega}{dt}, \quad (9 - 3)$$

$$\omega = p_p \Omega, \quad (9 - 4)$$

kde $\sigma = 1 - L_m^2/(L_1 L_2)$ (-) je tzv. rozptyl, $i_{1\alpha}$ (A) a $i_{1\beta}$ (A) jsou složky vektoru statorového proudu $\underline{i}_1^{\alpha\beta}$ (A), $\psi_{2\alpha}$ (Wb) a $\psi_{2\beta}$ (Wb) jsou složky vektoru rotorového magnetického toku $\underline{\psi}_2^{\alpha\beta}$ (Wb), $u_{1\alpha}$ (V) a $u_{1\beta}$ (V) jsou složky vektoru statorového napětí $\underline{u}_1^{\alpha\beta}$ (V), p_p (-) je počet polpárů stroje, ω (s^{-1}) je elektrická úhlová rychlosť hřídele, Ω (s^{-1}) je mechanická úhlová rychlosť hřídele, M (Nm) je vnitřní elektromechanický moment stroje a M_z (Nm) je moment zátěžný. [33]

9.2 I-n model asynchronního motoru

Jak již bylo v předcházejících částech zmíněno, pokud není k dispozici dostatečný počet LUTs pro výpočet kompletního matematického modelu, je možné využít PL na výpočet proudově-otáčkového, resp. *I-n* modelu a regulační procesy realizovat v PS.

Popis *I-n* modelu vychází ze základních rovnic, popisujících asynchronní motor, uvedených např. v [34] a [33] (rovnice jsou upraveny a přeznačeny dle moderních konvencí ale význam zůstává zachován). V teorii prostorových vektorů je možné tedy psát soustavu rovnic

$$\underline{u}_1^k = R_1 \underline{i}_1^k + \frac{d\underline{\psi}_1^k}{dt} + j\omega_k \underline{\psi}_1^k, \quad (9 - 5)$$

$$\underline{u}_2^k = R_2 \underline{i}_2^k + \frac{d\underline{\psi}_2^k}{dt} + j(\omega_k - \omega) \underline{\psi}_2^k, \quad (9 - 6)$$

$$\underline{\psi}_1^k = L_1 \underline{i}_1^k + L_m \underline{i}_2^k, \quad (9 - 7)$$

$$\underline{\psi}_2^k = L_2 \underline{i}_2^k + L_m \underline{i}_1^k, \quad (9 - 8)$$

kde \underline{u}_1^k (V) je prostorový vektor statorového napětí, \underline{u}_2^k (V) je prostorový vektor rotorového napětí, \underline{i}_1^k (A) je prostorový vektor statorového proudu, \underline{i}_2^k (A) je prostorový vektor rotorového proudu, $\underline{\psi}_1^k$ (Wb) je prostorový vektor magnetického toku statoru, $\underline{\psi}_2^k$ (Wb) je prostorový vektor magnetického toku rotoru, ω (s^{-1}) je elektrická úhlová rychlosť otáčení rotoru, ω_k (s^{-1}) je úhlová rychlosť otáčení použitého souřadnicového systému, R_1 (Ω), resp. R_2 (Ω) je rezistivita statorového, resp. rotorového vinutí, L_1 (H), resp. L_2 (H) je statorová, resp. rotorová indukčnost a L_m (H) je hlavní magnetizační indukčnost.

I-n model vychází z předpokladu, že otáčivá úhlová rychlosť souřadnicového systému $\omega_k = 0$ a tudy díž model je odvozován v souřadnicovém systému spojeným se statorem (souřadnicový systém $\alpha\beta$). Po vzolení daného souřadnicového systému pro soustavu rovnic platí

$$\underline{u_1^{\alpha\beta}} = R_1 \underline{i_1^{\alpha\beta}} + \frac{d\underline{\psi_1^{\alpha\beta}}}{dt}, \quad (9 - 9)$$

$$\underline{u_2^{\alpha\beta}} = R_2 \underline{i_2^{\alpha\beta}} + \frac{d\underline{\psi_2^{\alpha\beta}}}{dt} - j\omega \underline{\psi_2^{\alpha\beta}}, \quad (9 - 10)$$

$$\underline{\psi_1^{\alpha\beta}} = L_1 \underline{i_1^{\alpha\beta}} + L_m \underline{i_2^{\alpha\beta}}, \quad (9 - 11)$$

$$\underline{\psi_2^{\alpha\beta}} = L_2 \underline{i_2^{\alpha\beta}} + L_m \underline{i_1^{\alpha\beta}}. \quad (9 - 12)$$

V případě řízení asynchronního motoru s kotvou nakrátka orientovaného na rotorový tok je dále z rovnice 9 - 12 vyjádřen prostorový vektor $\underline{i_2^{\alpha\beta}}$, který je dále dosazen do upravené rovnice 9 - 10, u které je přepokládáno, že $\underline{u_2^{\alpha\beta}} = 0$.

Výsledná diferenciální rovnice pro prostorový vektor rotorového magnetického toku je

$$\frac{d\underline{\psi_2^{\alpha\beta}}}{dt} = \frac{R_2}{L_2} L_m \underline{i_1^{\alpha\beta}} + j\omega \underline{\psi_2^{\alpha\beta}} - \frac{R_2}{L_2} \underline{\psi_2^{\alpha\beta}}. \quad (9 - 13)$$

Rozepsáním představené diferenciální rovnice do reálné a imaginární složky vznikne soustava diferenciálních rovnic *I-n* modelu.

$$\begin{aligned} \frac{d\underline{\psi_{2\alpha}}}{dt} &= \frac{L_m R_2}{L_2} \underline{i_{1\alpha}} - \frac{R_2}{L_2} \underline{\psi_{2\alpha}} - \omega \underline{\psi_{2\beta}}, \\ \frac{d\underline{\psi_{2\beta}}}{dt} &= \frac{L_m R_2}{L_2} \underline{i_{1\beta}} - \frac{R_2}{L_2} \underline{\psi_{2\beta}} + \omega \underline{\psi_{2\alpha}}. \end{aligned} \quad (9 - 14)$$

10 Použité nástroje pro vývoj aplikace pro PS a PL

V této části jsou představeny jednotlivé nástroje, využívané při tvorbě programu pro PS a akcelerované aplikace (kernelu) realizované na PL. Je důležité zmínit, že v PS je skutečně spouštěn zkompilovaný program vytvářený pomocí jazyka C, C++ nebo Python. Na PL je ovšem vytvořen HW, který reprezentuje myšlené algoritmy aplikace. Tento HW je popisován pomocí nízkoúrovňových jazyků, do kterých je algoritmus převeden pomocí HLS z jazyka C. Není tudíž korektně správné mluvit o tom, že se vytváří program pro FPGA. Z toho důvodu bude v této práci používáno označení pro vytváření HW na PL *vytváření kernelu* (creation of the kernel). Označení *kernel* je myšleno v odlišném významu než je využíváno v části *RealTime Linux Patch*.

Tvorba akcelerované aplikace může být obecně prováděna více způsoby. Tento způsob závisí na použitém vývojovém nástroji pro daný HW. V této práci je využíváno SOM od firmy Xilinx, proto je výhodné využívat již připravené nástroje, které umožní snazší vývoj SW, tvorbu HW a přípravu systému na PS.

V této práci veškerý používaný SW od firmy Xilinx je po registraci volně dostupný ke stažení na [35].

10.1 Xilinx Vivado

Xilinx Vivado je nástroj, používaný pro tvorbu HW architektury, resp. platformy, pro kterou bude v další části postupu možné vytvořit akcelerovanou aplikaci. Ve Vivado je možné tvořit HW návrh, převeditelný do HDL, který bude spustitelný v PL bez použití Vitis HLS. Pro vývojáře HW na FPGA může sloužit i jako hlavní vývojářský nástroj.

Se znalostí VHDL je možné ve Vivado vytvářet požadovaný HW design, ovšem tvorba designů ve Vivado s využitím VHDL je relativně náročnou záležitostí a není předmětem této práce. Využití VHDL je však nevyhnutelnou součástí budoucího výzkumu využití těchto platforem pro řízení elektrických polohonů.

Xilinx Vivado je součástí instalačního balíčku *Xilinx Unified Installer*, dostupného z [35]. Součástí balíčku verze 2022.2 SFD je nástroj *Xilinx Vitis*. Je doporučeno instalovat oba tyto programy a vyvarovat se oddělené instalace, jež může přinášet problémy se vzájemnou i zpětnou kompatibilitou jednotlivých nástrojů.

10.2 Xilinx Vitis

Xilinx Vitis je nástroj, který slouží k vytváření akcelerovaných aplikací na zařízení firmy Xilinx. Tento nástroj obsahuje základní vrstvu s názvem Xilinx Vitis HLS, která slouží jako jádro převodu vytvářených aplikací v C, C++ a OpenCL do RTL. V programu Xilinx Vitis bude vytvářena největší část aplikace, proto je vhodné nastítit postup, jakým Vitis pracuje.

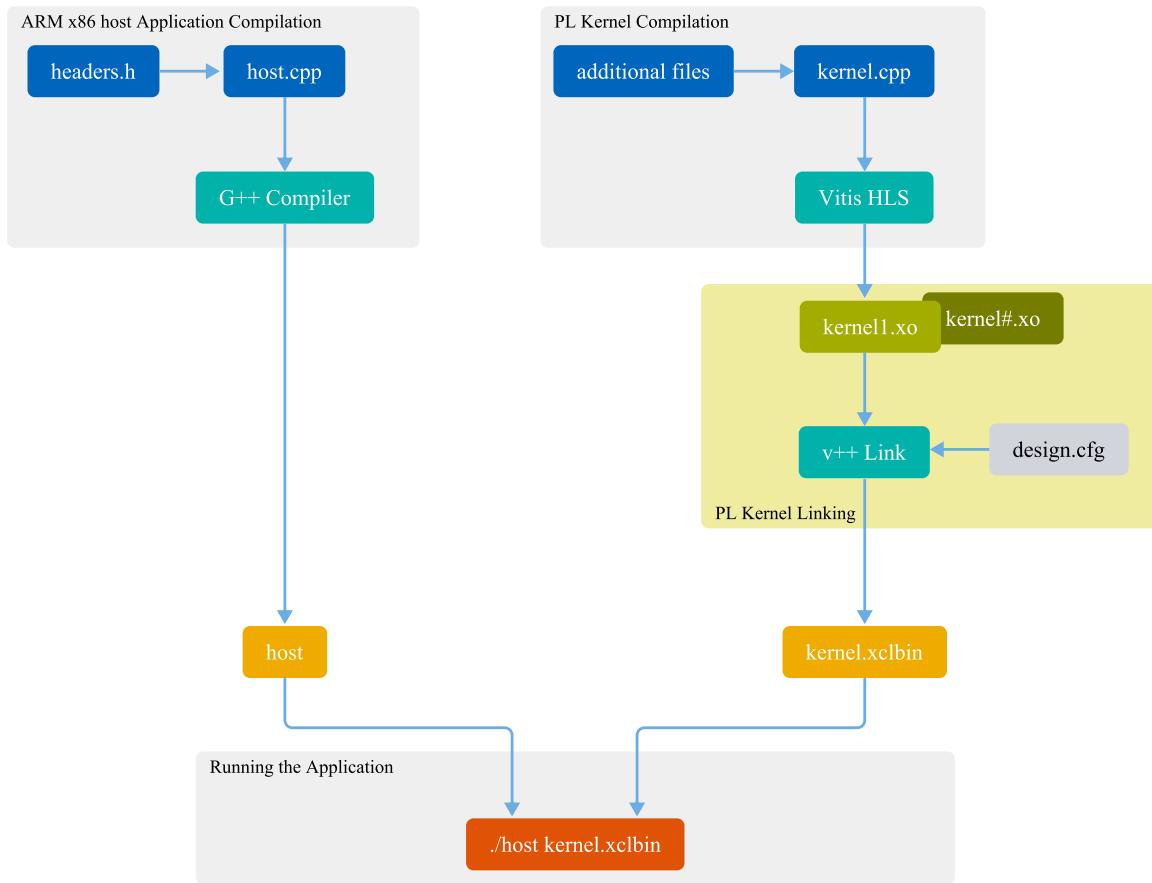
Nejprve je vytvořen tzv. „host program“ bežící na PS, který je vyvíjen v C/C++ jazyku (popř. Python), používající Xilinx Runtime (XRT) Application Programming Interface (API). Tento program je následně komplikován pomocí g++ kompilátoru, který vytvoří spustitelný soubor pro procesor. Tento host program komunikuje s akcelerovanou částí aplikace (kernel), umístěnou v PL. Blok *ARM x86 host Application Compilation* na obr. 10 - 1 naznačuje, jakým způsobem je vytvářena aplikace pro host procesor. [27]

Poté Vitis HLS compiler přeloží C/C++ zdrojový kód pro kernel do register transfer level (RTL) (úrovně registrů). Produkty této komplikace mají příponu „.xo“ (Xilinx Object) a mohou být spojovány do binárního souboru s příponou „.xclbin“ pomocí Vitis linkeru. Souborem *kernel.xclbin* je poté možné nakonfigurovat PL. [27]

Na obr. 10 - 1 je blokově znázorněn postup tvorby spustitelné aplikace v programu Vitis. Tento dia-

gram předpokládá, že již byla vytvořena HW platforma ve Vivado a *PetaLinux* systém.

Produkty větví tvorby programu pro PS a konfigurace pro PL je po jejich dokončení možné použít v daném heterogenním systému. Host program zařídí nakonfigurování PL pomocí souboru *kernel.xclbin* a následné zpracování výsledků. Blok s názvem *Running the Application* je vykonáván v prostředí *PetaLinux* v simulátoru (QEMU) nebo na fyzickém zařízení (vývojová deska).



Obr. 10 - 1 Blokový diagram tvorby spustitelné aplikace v prostředí Vitis. (převzato z [27], upraveno)

10.3 PetaLinux Tools

PetaLinux Tools je nástroj, který slouží k vytvoření systému *PetaLinux*, jež bude spuštěn na PS v daném SOM nebo SoC. Z tohoto systému je poté možné spouštět navazující programy host (na PS) a kernel (na PL).

PetaLinux systém je možné nakonfigurovat dle požadavků aplikace. Při tvorbě tohoto systému je možné konfigurovat jádro systému (kernel), balíčky, které budou do systému nainstalovány, vytvořit uživatele systému nebo vybrat, kde v paměti bude systém umístěn (RAM, SD karta apod.). [36]

V případě tvorby systému, který je konfigurovaný uživatelem, je třeba postupovat obezřetně a dodržovat nastavené postupy konfigurace, protože v případě chyby je nutné překonfigurovat chybnou část nebo někdy kompletní systém. Tvorba *PetaLinux* systému je časově náročný postup, u něhož je problematický debugging.

Pro funkční instalaci *PetaLinux Tools* je nutné mít v systému, kde bude docházet k tvorbě *PetaLinux* systému, nainstalované správné verze systémových a aplikačních balíků, které jsou nutnou prerekvizitou tvorby *PetaLinux*. Požadavky na balíky je možné nalézt při nahlédnutí do dokumentace [37] instalované

verze *PetaLinux Tools*. V dokumentaci v sekci *Installation Requirements* se nachází odkaz označený *PetaLinux <version> Release Notes*, který ve spodní části obsahuje stáhnutelný soubor *<version>_PetaLinux_Package_List.xlsx*, jež obsahuje seznam požadovaných balíků a jejich verze. Bez použití podporovaných balíků by nepracoval nástroj *PetaLinux Tools* správně.

10.4 RealTime Linux Patch

Protože operační systém Linux nebyl původně navrhován pro využití v embedded systémech, ale v obecných zařízeních jako jsou servery a stolní počítače, nebyl tento systém vhodný pro řešení úloh v reálném čase (real time, RT). Proto se objevila snaha upravit tento systém takovým způsobem, aby jej bylo možné v RT systémech využívat.

Real time systémy je dle [38] možné rozdělit do jednotlivých úrovní podle časových požadavků řízeného systému v reálném čase na:

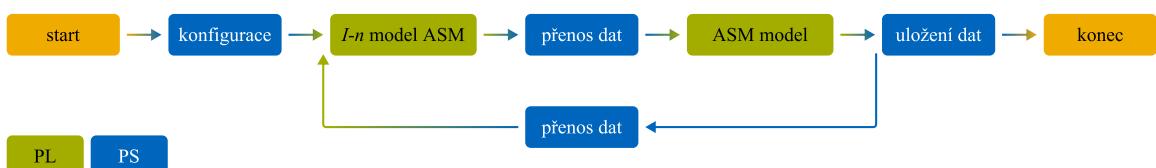
- **Soft Real Time** – aplikace, ve které je hlavním parametrem kvalita výsledků, pokud v některých případech nedojde k dodržení časových omezení jednotlivých úkonů, nemá tato chyba vliv na zdraví člověka nebo stav majetku,
- **Firm Real Time** – pokud v aplikaci nedojde k dodržení časových omezení výpočtu, je výsledek daného výpočtu považován za neplatný a nelze jej použít,
- **Hard Real Time** – v aplikaci je zakázáno nedodržení časových omezení, kdyby došlo k překročení pevně daných časových rámců, může vzniklá situace vést k ohrožení lidských životů nebo stavu majetku.

V [38] jsou představeny původní přístupy, kdy pro dodržení časových omezení a tzv. „*preemptibility*“ (přerušitelnosti vykonávaného vlákna) byl využit *cokernel*.

Moderní způsob spočívá v aplikování Linux patch pro danou verzi kernelu (pojmeme kernel v tomto případě není myšlena akcelerovaná aplikace, ale jádro operačního systému Linux), kdy není přidávána do systému další vrstva jádra, ale původní jádro je upravováno. Úpravy spočívají ve změně některých způsobů funkce jádra a přerušení. Tento patch se obecně nazývá *PREEMPT_RT* a o začlenění jeho principů do mainline kernelu je dlouhodobě usilováno. [38]

Popis state-of-art *PREEMPT_RT* je popsán v [38].

V této diplomové práci je patch využit pro získání co největší přerušitelnosti jádra, tudíž aby byl kernel *FULLY PREEMPTIBLE*. Pokud by tomu tak nebylo, nebyly by výsledky simulací matematických modelů při použití PL a PS konzistentní. Pokud je využívána architektura simulace, naznačená na obr. 10 - 2, dojde při opakovaném spuštění aplikace s vysokou pravděpodobností k získání znehodnocených výsledků, které není možné použít. Při spuštění aplikace se tento problém projeví nevalidními výsledky, které neodpovídají žádnému ze zadaných parametrů. Po několikanásobném spuštění aplikace je možné získat validní výsledky, ovšem četnost, kdy dochází k získání nevalidních výsledků, je značně vysoká.



Obr. 10 - 2 Graf prováděné simulace při testování *PREEMPT_RT* Linux Patch.

10.4.1 Postup aplikace PREEMPT_RT patch

Patch je možné aplikovat několika způsoby. V této práci byl aplikován při tvoření *PetaLinux* systému pomocí úpravy konfiguračních souborů build procesu.

Pro bezproblémové aplikování patche je vhodné nejdříve vytvořit *build PetaLinux* systému s minimální konfigurací a bez aplikovanání patch souboru a až po úspěšném vytvoření systému patch aplikovat a build proces opakovat. Pro funkční aplikaci patch souboru je nutné znát verzi jádra *PetaLinux*, na který bude aplikován. Označení verze je možné získat z **Makefile** souboru umístěného v cestě naznačené v kódu 10 - 1, kde <petalinux-project> je označení pro kořenový (root) adresář *PetaLinux* projektu.

```
1 <petalinux-project>/build/tmp/work-shared/xilinx-k26-kr/kernel-source/
  Makefile
```

Kód 10 - 1 Cesta Makefile souboru, ze kterého je možné získat označení verze jádra systému PetaLinux.

Protože je zmiňovaný **Makefile** soubor rozsáhlý a pro určení verze kernelu je signifikantní pouze jeho úvodní část, je v kódu č. 10 - 2 vynechána podstatná část souboru, která není pro aplikování patche podstatná.

```
1 # SPDX-License-Identifier: GPL-2.0
2 VERSION = 5
3 PATCHLEVEL = 15
4 SUBLEVEL = 36
5 EXTRAVERSION =
6 NAME = Trick or Treat
7 ...
```

Kód 10 - 2 Významná část Makefile souboru pro určení verze jádra PetaLinux systému.

Z kódu 10 - 2 je možné vyčíst, že je třeba využít patch pro verzi jádra 5.15.36. Pokud není v souboru informace ohledně verze jádra linuxu uvedena, je možné vytvořit *PetaLinux* obvyklým způsobem, vytvořit obraz systému, ten nahrát na SD kartu, provést spuštění systému na vývojové desce a po úspěšném přihlášení do systému vyvolat příkaz **uname -a** a dle uvedených informací odvodit verzi jádra.

Poté je z adresy <https://cdn.kernel.org/pub/linux/kernel/projects/rt> možné stáhnout patch pro zjištěnou verzi jádra. V této práci byl využit patch jádra pro *PetaLinux* 2022.2 umístěný v cestě 5.15/older/patch-5.15.36-rt41.patch.gz.

Dalším krokem je extrahovaný soubor patch-5.15.36-rt41.patch přenést do složky <petalinux-project>/project-spec/meta-user/recipes-kernel/linux/linux-xlnx/. Build proces musí následně pracovat s informací o umístění patch souboru, proto je vyžadováno aby do konfiguračního souboru

```
<petalinux-project>/project-spec/meta-user/recipes-kernel/linux/linux-
xlnx_%.bbappend byl na poslední řádek zapsán příkaz
SRC_URI:append = "file://patch-5.15.36-rt41.patch",
kde patch-5.15.36-rt41.patch je název patch souboru. Ukázka linux-xlnx_%.bbappend souboru, využitého v této práci, je v kódu 10 - 3. Jak je vidět, soubor obsahuje informace o různých konfiguračních souborech pro tvorbu jádra Linux systému.
```

```
1 FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
2
```

```

3 SRC_URI:append = " file://bsp.cfg"
4 KERNEL_FEATURES:append = " bsp.cfg"
5 SRC_URI:append = " file://patch-5.15.36-rt41.patch"
6 SRC_URI += " file://user_2023-04-07-11-32-00.cfg"

```

Kód 10 - 3 Ukázka konfiguračního souboru pro aplikování Linux patch souboru.

Aby došlo k aplikování změn, vyvolaných RT patch souborem, je nutné během build procesu provést určité změny v konfiguraci vytvářeného *PetaLinux* projektu. (postup předpokládá, že již byl vytvořen první build s minimální konfigurací) Opět jsou v *PetaLinux Environment* (prostředí) vyvolány příkazy **petalinux-config** pro první konfiguraci HW a **petalinux-config -c kernel** pro konfiguraci jádra. Po otevření konfigurační nabídky jádra je nutné provést změny, vyznačené v 10 - 4.

```

1 General setup -> Timers subsystem -> High Resolution Timer Support <*>
2 General setup -> Preemption Mode -> Fully Preemptive Kernel (RT) <*>
3 Main menu -> Kernel Features -> Timer frequenc -> 1000 Hz <*>
4 Main menu -> CPU power Management -> CPU Frequency Scaling < >

```

Kód 10 - 4 Úpravy v konfiguraci jádra pro RT patch.

Značení:

- < > funkce není aktivována,
- <*> funkce je aktivována.

Po konfiguraci je již možné provádět build proces klasickým způsobem, popsaným v části *Tvorba PetaLinux*.

Představený postup čerpá informace o aplikování patch souboru z [39], [40] a z experimentálního zjištění autora.

10.5 Programovací prostředí – operační systém Linux

Pro práci s představenými nástroji *Xilinx Vivado*, *Xilinx Vitis* a *PetaLinux Tools* je nutné využívat podporovaných operačních systémů.

Požadavky na operační systémy je možné nalézt na stránkách dokumentace <https://docs.xilinx.com>. V době zpracování této práce jsou pro nejnovější verze nástrojů 2022.2 požadavky pro Xilinx Vivado dostupné v [41]. Požadavky na operační systém pro Xilinx Vitis v [27]. Pro využívání a tvorbu PetaLinux Tools je třeba dodržet systémové požadavky uvedené v [37].

Pokud uživatel využívá starších verzí vývojových nástrojů, je doporučeno využít operační systém Linux. Pro tuto práci byl nejdříve využíván systém Ubuntu 18.04 LTS (Bionic Beaver), dostupný ke stažení na adrese <http://old-releases.ubuntu.com>. V průběhu práce došlo k aktualizování verzí vývojových nástrojů, které byly původně kompatibilní pouze s verzí Ubuntu 18.04 a nižší. Veškerá práce a postupy byly po aktualizaci přeneseny na novější verzi systému Ubuntu 20.04 LTS (Focal Fossa).

Je důležité poznamenat, že když Vivado podporuje některou z novějších verzí Ubuntu, není jisté, že jí podporuje také *PetaLinux Tools*. Vždy je doporučeno využívat starší verze a kontrolovat vzájemnou kompatibilitu, aby se předešlo zbytečné ztrátě času při reinstalaci nástrojů.

V případě využívání představených nástrojů a systému Linux je třeba dbát na správné postupy instalací a v případě problémů využívat dostupné dokumentace.

11 Struktura složek

Aby byl vývoj, debugging, deployment a verzování aplikace co nejméně problematickým a zdlouhavým procesem pro vývojáře (jak HW tak SW), je vhodné zavést pro daný projekt pevný systém složek (file system), který bude dodržován napříč projekty. V případě existence takového systému je možné vytvořit postupy a skripty, které značně urychlí práci na vyvýjeném projektu.

Tyto skripty mohou sloužit ke snadnějšímu přenosu souborů mezi jednotlivými složkami pro potřeby daných vývojových nástrojů, přenos souborů na vývojovou desku a nebo k výrazně rychlejší práci s vývojovými nástroji *PetaLinux Tools* a Vitis IDE. V této práci bude dodržována struktura naznačená v kódu 11 - 1.

```
1 - projects folder
2   - top folder (project name)
3     - transfer           // user generated
4     - hw                 // vivado project
5     - petalinux          // petalinux project
6     - linux-files        // user generated folders
7       - pfm
8         - boot
9         - sd_dir
10      - dtg_out           // created when converting device tree from XSA file
11      - sysroots          // created by ./sdk.sh -d ../../linux-files
12    - vitis
```

Kód 11 - I Struktura složek, využívaná při tvorbě projektů k dosažení lepšího DX.

Tato struktura přináší možnosti rychlejšího pohybu v projektu pomocí vzdáleného přístupu ssh a emulátoru terminálu, který umožnuje provádět build aplikace i bez použití GUI Vitis IDE. Využíváním headless módu dochází k odstranění některých nedostatků SW. Ovšem GUI je vhodné na provádění úkonů, jejichž způsob provedení v headless módu nebyl při realizaci této práce objeven (tvorba platformy, tvorba aplikace, automatické vytváření *makefile* souborů apod.).

V případě verzování projektu je ovšem důležité si uvědomit, že některé soubory mají značnou velikost a některé složky obsahují velmi mnoho souborů (více než 8 000 souborů). Proto je nutné tuto skutečnost vnímat a dle vlastních požadavků vyjmout vybrané prvky z verzování.

12 Tvorba HW architektury Xilinx Vivado

Aby bylo možné vytvořit akcelerovanou aplikaci ve Vitis s pomocí HLS C++, je třeba připravit platformu, resp. hardware, pro který bude daná aplikace vyvíjena. K tvorbě platformy je využit Xilinx Vivado. V tomto programu je možné konfigurovat jednotlivé IP (intellectual property) prvky jako je ZynQ jednotka, GPIO, Timer, SPI komunikace a další. Výsledkem tvorby platformy v této práci je soubor XSA, který je použit pro konfiguraci *PetaLinux* systému a slouží jako vstupní informace pro tvorbu Platformy ve Vitis. Ve Vivado je možné vytvářet aplikace přímo v VHDL.

Tvorba HW pro různé platformy (Digilent Zybo, Xilinx Kria KR260, SoC, SOM) má částečně odlišné specifikace a odlišný postup. Rámcový postup je však totožný pro většinu platem využívající zařízení od firmy Xilinx, Inc.

V této sekci bude popsána tvorba platformy pro vývojovou desku Xilinx Kria KR260 Starter Kit, na níž byla realizována finální aplikace. V příloze práce je naznačen postup tvorby základní platformy pro vývojovou desku Digilent Zybo.

12.1 Vivado Board Files

Aby bylo možné snadněji vytvořit potřebnou HW architekturu, firmy často dodávají ke svému produktu *Board Files* soubory, obsahující přednastavení, konfigurace, informace a způsob připojení IP bloků k reálným součástím (constraints). [42]

Samozřejmě by bylo možné HW architekturu vytvořit i bez těchto konfiguračních souborů, ovšem postup tvorby by byl značně náročnější. Pro vývojovou desku Xilinx Kria KR260 Starter Kit výrobce dodává Board files již s instalací Vivado. Pro používanou vývojovou desku Digilent Zybo Zynq-7000 je možné stáhnout tyto soubory z [42]. Způsob instalace board files je popsán v oficiální dokumentaci firmy Digilent, Inc. v [43].

Po úspěšné instalaci souborů je možné spustit Xilinx Vivado a vytvořit potřebnou HW architekturu pro akcelerovanou aplikaci.

12.2 Tvorba HW designu pro Xilinx Kria KR260 vývojovou desku

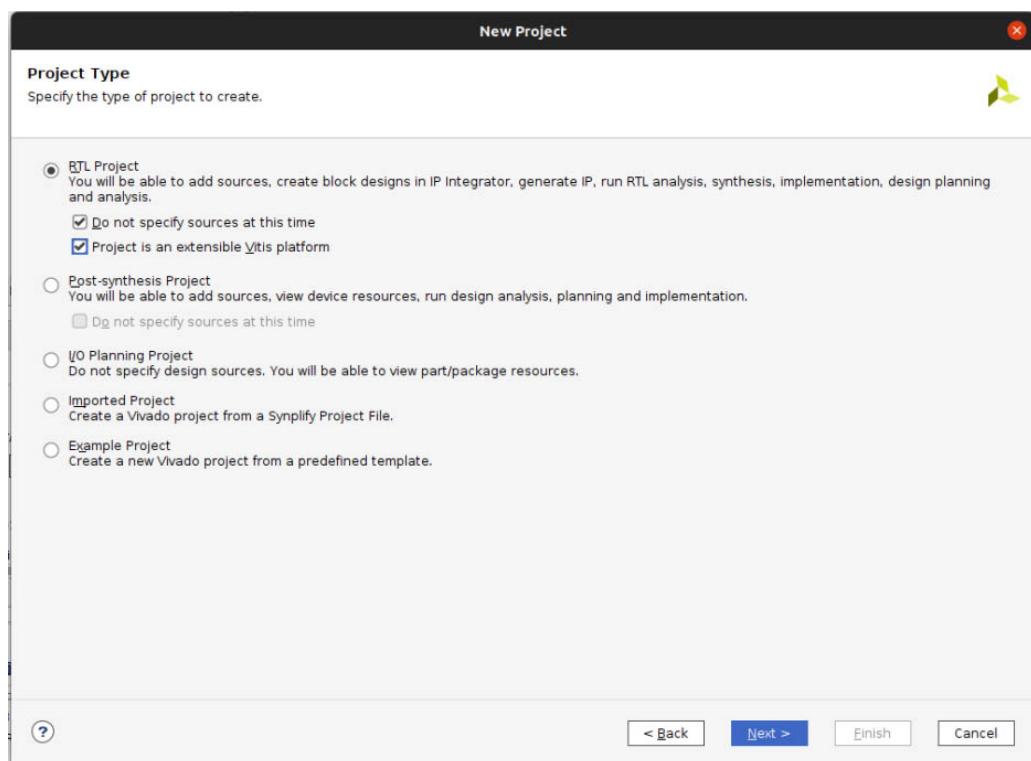
Při tvorbě designu pro vývojovou desku Xilinx Kria KR260, byly čerpány základní rámcové informace o postupu z [44], [45] a [46]. Konkrétní postup se liší dle vytvářené aplikace a zkoumaných vlastností.

Protože první zkoumání využitelnosti SoC bylo prováděno na desce Digilent Zybo Zynq-7000, je v příloze *Tvorba HW designu pro Digilent Zybo Zynq-7000* nastíněn postup tvorby HW designu pro původní desku. Konfigurace PS a tvorba HW pro Digilent Zybo se odlišuje převážně proto, že Zybo používá starší PS Zynq-7000, oproti novějšímu PS MPSoC Zynq UltraScale+ v Xilinx Kria.

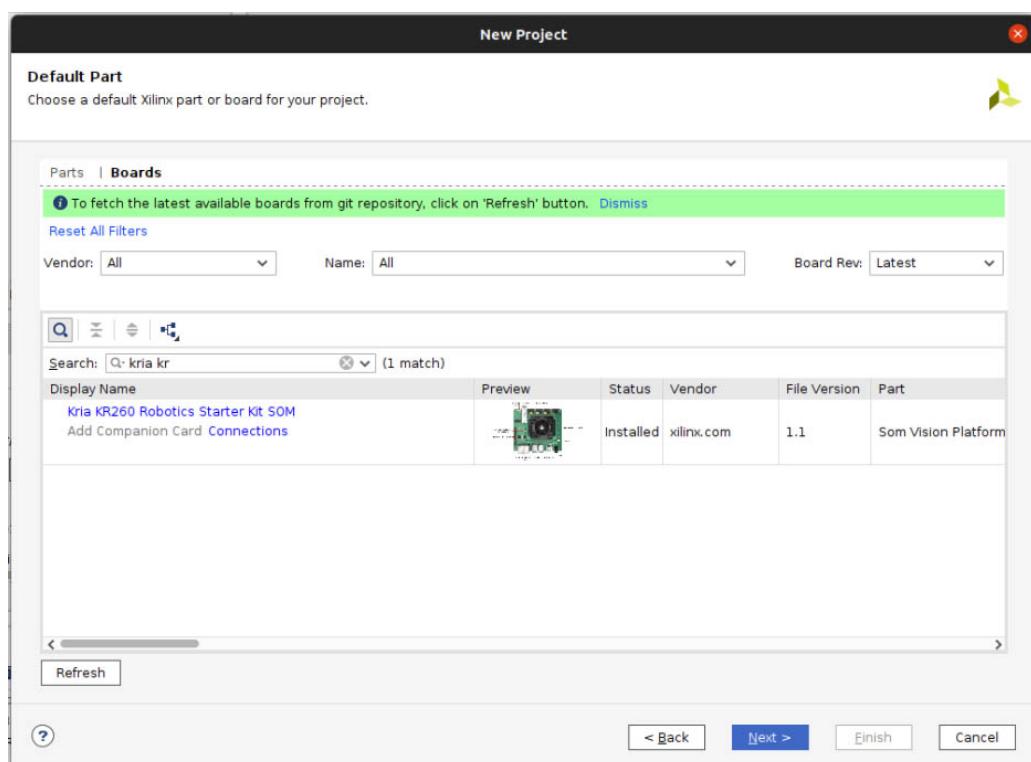
Prvním krokem je vytvoření Vivado projektu a jeho umístění do složky hw (popis struktury složek v projektu je představen v části *Struktura složek*).

Postup tvorby projektu začíná pro většinu akcelerovaných aplikací stejným způsobem. Po otevření programu Vivado stačí vytvořit nový project typu *RTL Project* a aktivovat nastavení *Project is an extensible Vitis platform*. Ukázka nabídky tvorby projektu je na obr. 12 - 1.

Dalším krokem při zakládání projektu je zvolení prvku, pro který bude vyvíjený HW design určen. Je možné zvolit přímo komponentu, nebo již přednastavené vývojové desky. Pokud není vývojová deska v repozitáři od Xilinx, je možné ji vložit dle způsobu popsáного v části *Vivado Board Files*. Xilinx Kria KR260 je však již součástí daného repozitáře a je možné ji v repozitáři vyhledat a zvolit. Výběr desky z repozitáře je zobrazen na obr. 12 - 2.

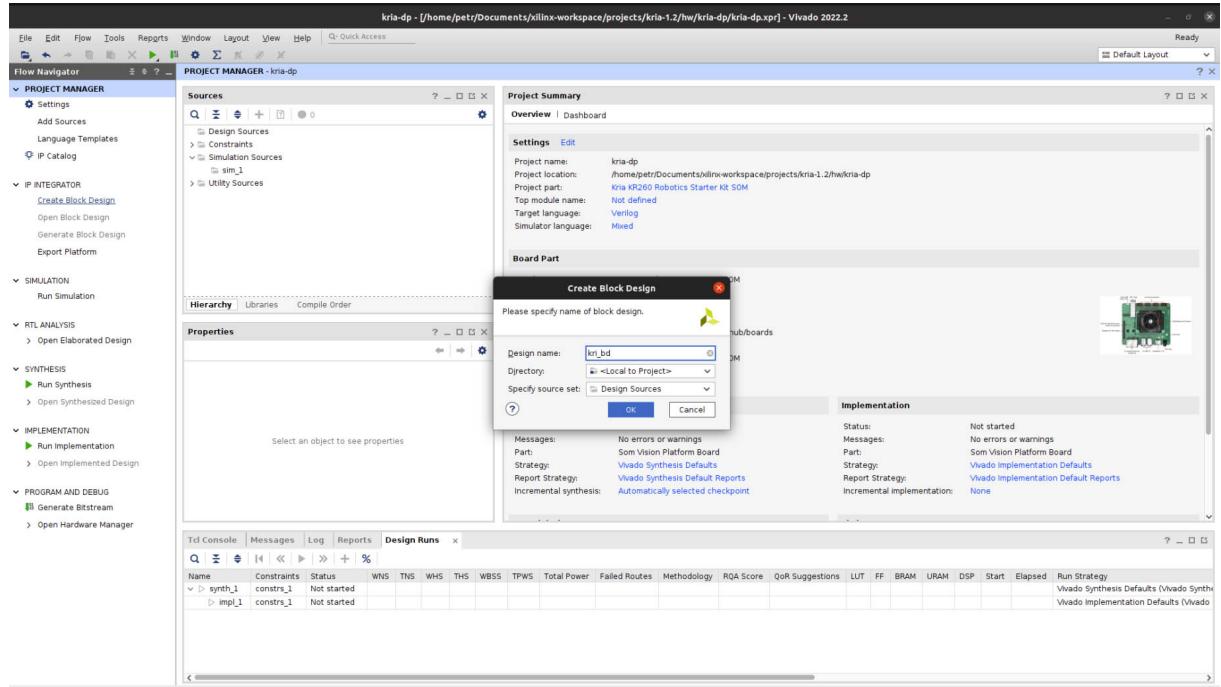


Obr. 12 - 1 Xilinx Vivado – volba typu projektu pro Xilinx KR26, použitelného dále jako platforma ve Vitis.



Obr. 12 - 2 Xilinx Vivado – výběr základní komponenty, pro který bude HW design vytvářen.

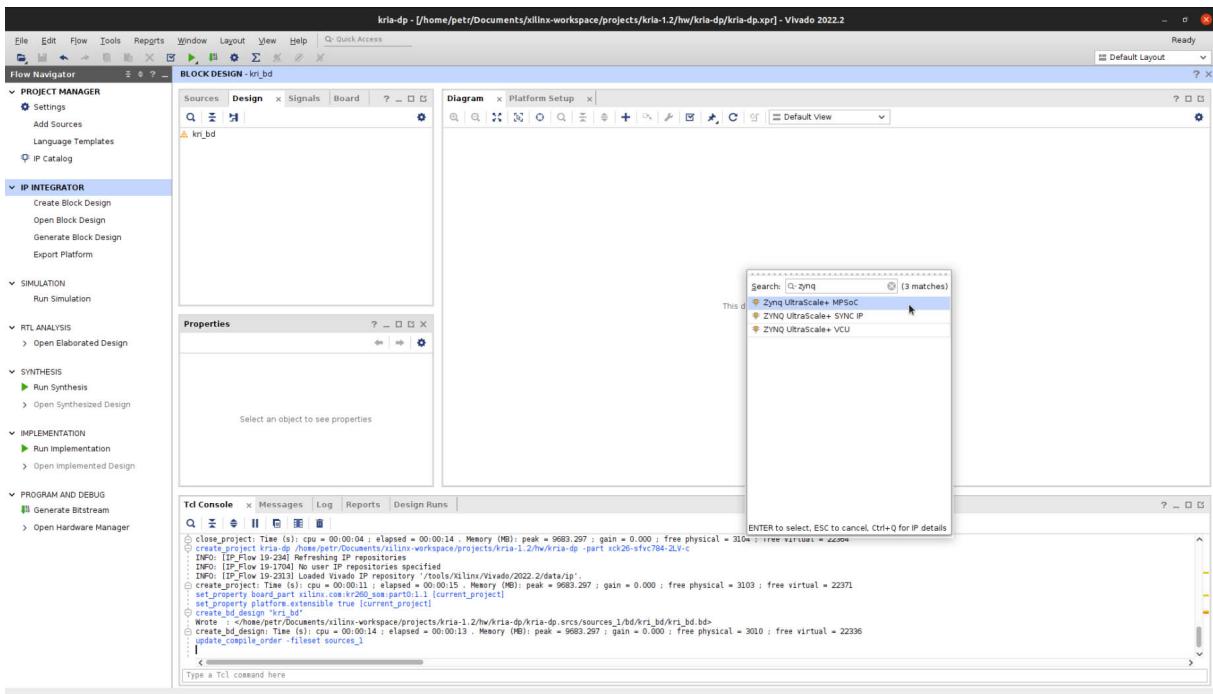
Po vytvoření projektu je uživateli zobrazena hlavní Vivado obrazovka. V základním nastavení jsou v pravé části obrazovky zobrazeny informace o vybrané základní komponentě/desce a v levé části pracovní menu. Pro pokračování ve tvorbě designu je třeba zvolit v menu odkaz *Create block design* a pojmenovat jej dle požadavků autora designu. Menu a nabídka vytváření blokového designu je zobrazena na obr. 12 - 3.



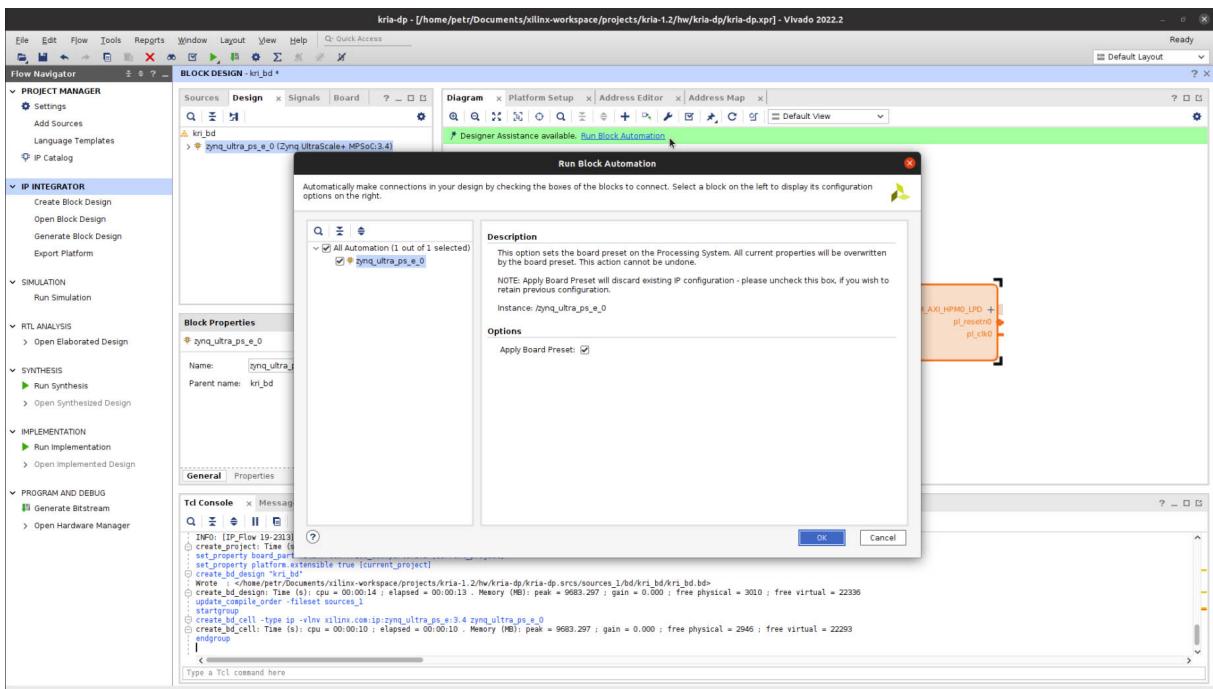
Obr. 12 - 3 Xilinx Vivado – nabídka vytváření block design.

Až po krok vytváření block designu byl postup velmi podobný pro obě představené vývojové desky. Nyní je již možné přistoupit k vlastní tvorbě blokového designu v kartě *Diagram*.

Bloky lze přidávat znakem „+“ v aktivním okně nebo po kliknutí pravého tlačítka myši do volného prostoru v téže okně a zvolení *Add IP*. Prvním krokem je přidání PS, v případě Xilinx KR26 se jedná o IP s názvem **Zynq UltraScale+ MPSoC**. Výhoda používání Vivado je taková, že po vložení některých bloků je k dispozici aktivace automatického propojení/nastavení některých bloků IP. Po vložení PS bloku je vhodné tuto automatizaci spustit pomocí aktivního odkazu, zobrazeného na kartě *Diagram* v obr. 12 - 5.

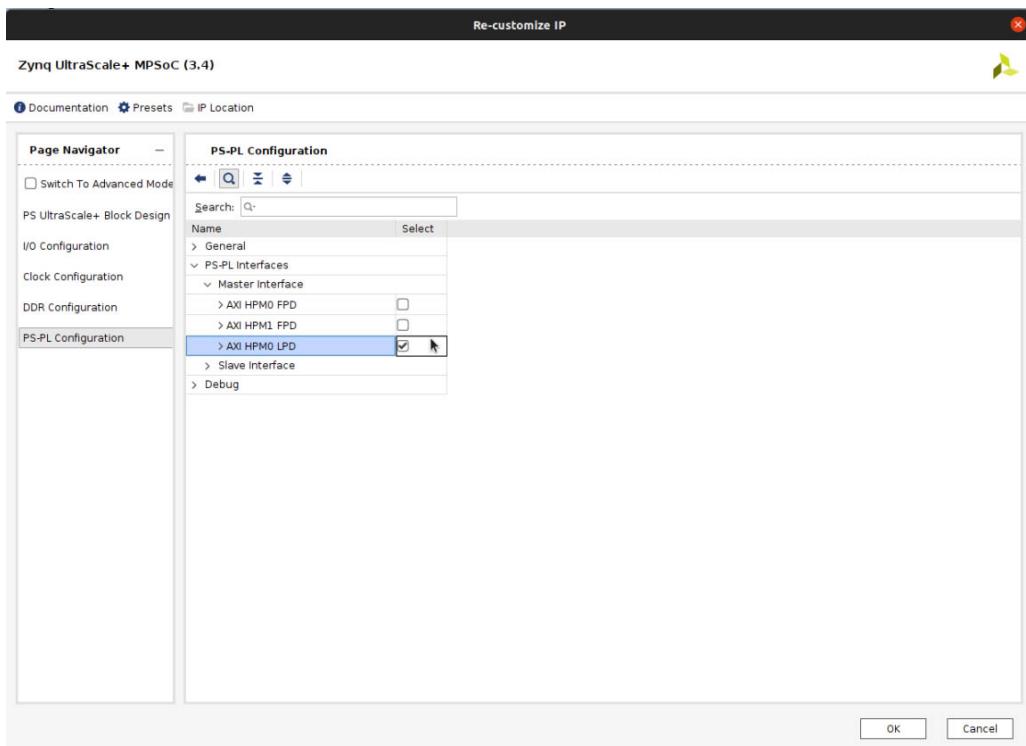


Obr. 12 - 4 Xilinx Vivado – vložení PS IP bloku.



Obr. 12 - 5 Xilinx Vivado – automatické propojení pro PS.

Další krok je důležitý pro tvorbu akcelerované aplikace pomocí Vitis. Je třeba předkonfigurovat PS AXI (rozhraní) takovým způsobem, aby AXI s vysokým výkonem bylo využito až pro akcelerovanou aplikaci a nikoliv v některém z automatických propojení, jejichž pozitivní přínos byl představen v předchozím odstavci. Tato informace je popsána v [46] ale také v oficiálních repozitářích [47] (v sekci *Add Interrupt Support, krok 1*) pro tvorbu platformy vývojové desky Xilinx Kria KV260, jež používá totožný modul Xilinx Kria K26. Obr. 12 - 6 obsahuje ukázku nastavení bloku **Zynq UltraScale+ MPSoC** a daných rozhraní.



Obr. 12 - 6 Xilinx Vivado – zablokování FPD a odblokování LPD pro block design.

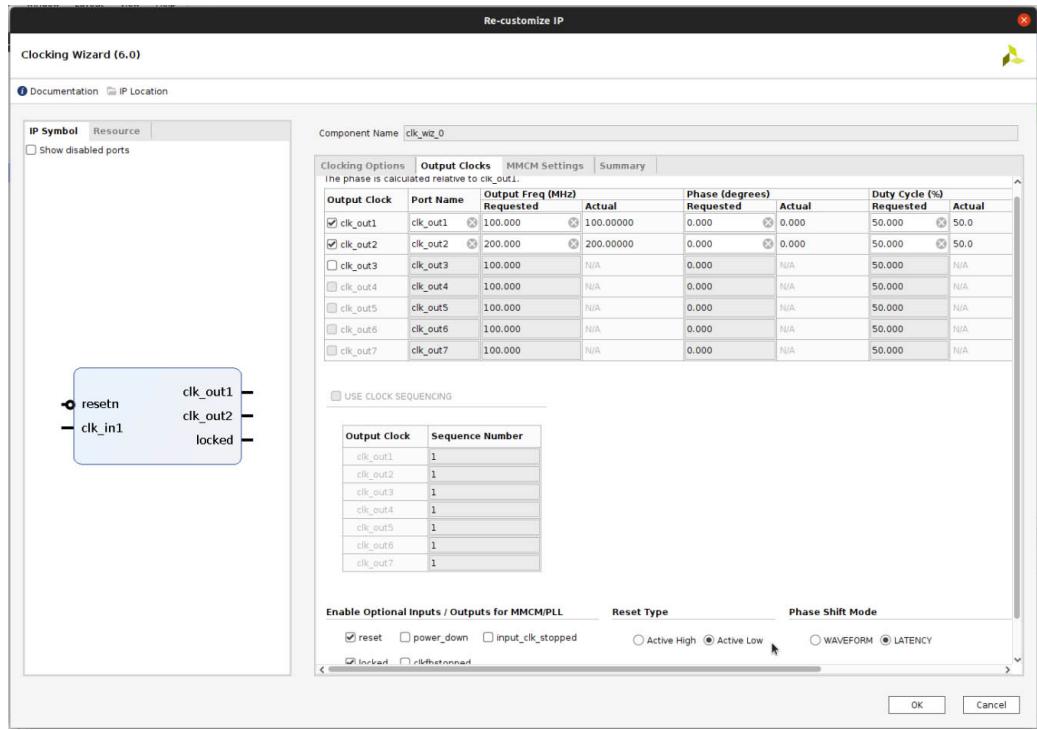
Protože dle [47] je počet clock signálů `p1_clk` z PS omezený, je vhodné pro vytvoření požadovaných taktovacích signálů (pro PL) využít blok **Clocking wizard**. V tomto bloku je možné vytvořit taktovací signály s požadovanými parametry. Pokud počet signálů nestací, je možné přidat další IP.

Pro design ukázkové platformy v této práci jsou využity tři taktovací signály s frekvencí 100, 200 a 20 MHz. Také je důležité nastavit *Reset type* na *Active Low*. Příklad nastavení bloku **Clocking wizard** je na obr. 12 - 7.

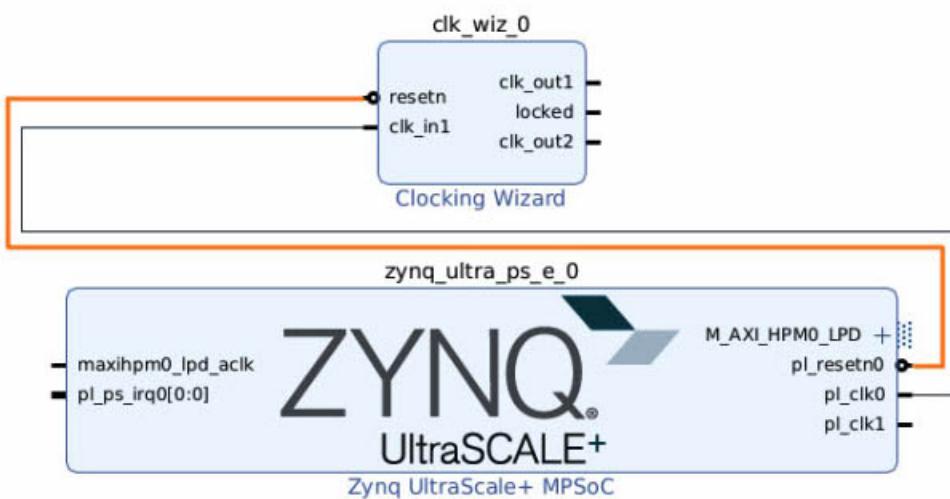
Po nastavení požadovaných taktovacích signálů je možné opět pomocí aktivního odkazu automatizace spustit automatické propojení bloků. Po úspěšném provedení předchozích konfiguračních kroků a automatizace je získáno schéma blokového designu na obr. 12 - 8.

K bloku **Clocking Wizard** je nyní vhodné připojit bloky **Processor System Reset** dle obr. 12 - 9.

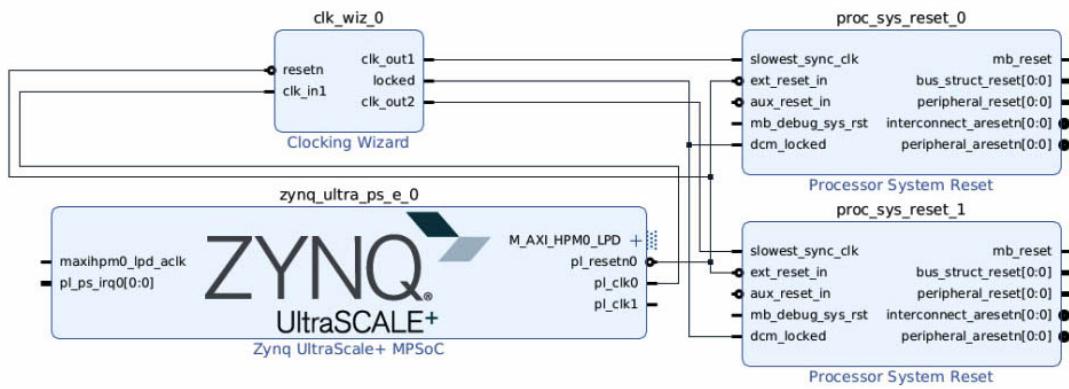
Vstup pro přerušení `p1_ps_irq` může obsahovat maximálně 16 interrupt signálů. Pro design v této aplikaci se jedná o dostačující počet, ovšem pokud je vyžadováno, aby aplikace využívala více signálů přerušení, je třeba využít blok **AXI Interrupt Controller**. Doporučené nastavení tohoto IP je uvedeno na obr. 12 - 10. Aby bylo možné připojit signál k `p1_ps_irq` je nutné přepnout nastavení *Processor Interrupt Type and Connection -> Interrupt Output Connection* na *Single* z výchozího *Bus*.



Obr. 12 - 7 Xilinx Vivado – nastavení bloku Clocking wizard.



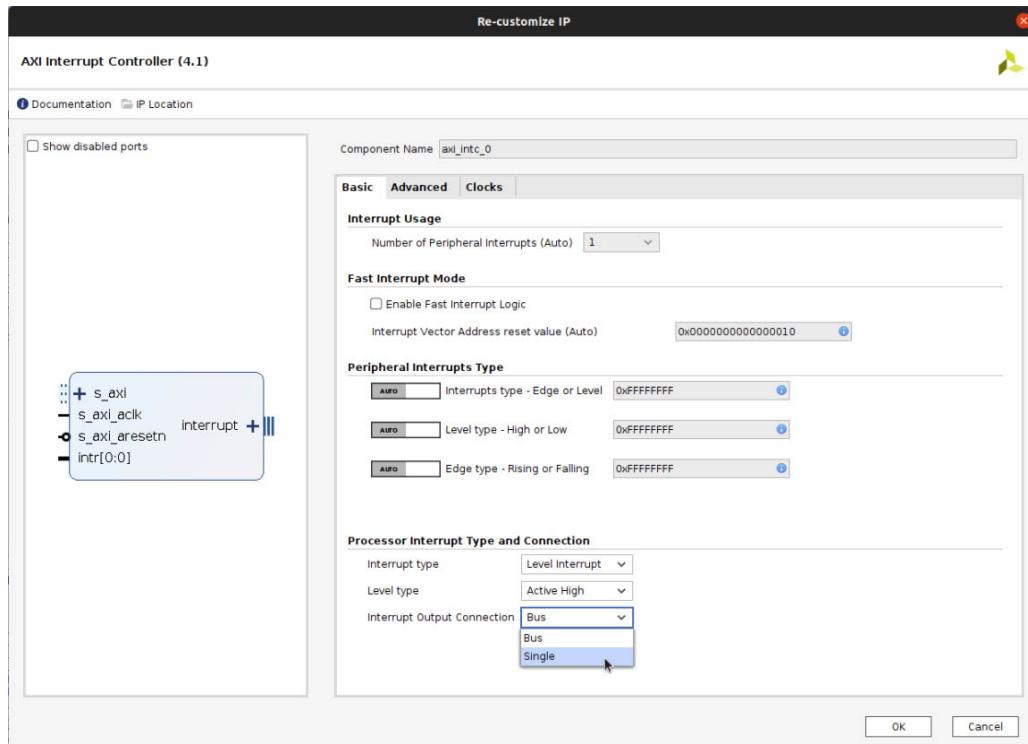
Obr. 12 - 8 Xilinx Vivado – blokový design s PS a blokem Clocking wizard po provedení automatizace.



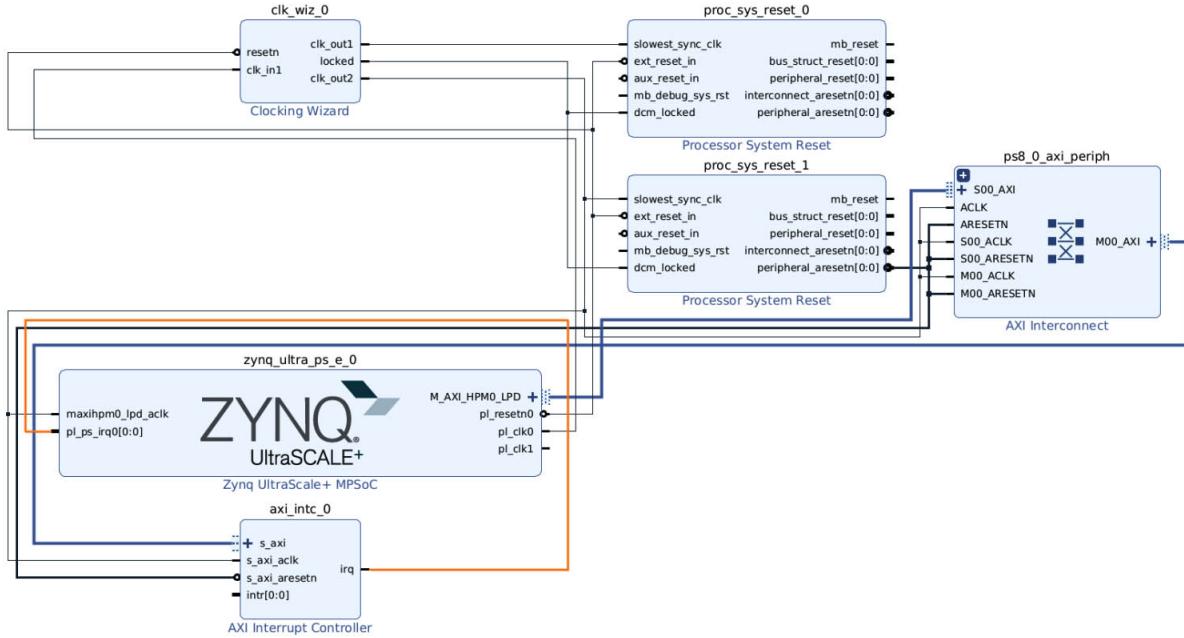
Obr. 12 - 9 Xilinx Vivado – propojení bloků Clocking wizard a Processor System Reset.

V případě nevyužití bloku AXI Interrupt Controller, nebo při snaze přivést signály přerušení přímo do p1_ps_irq vstupu PS systému, je pro připojení více signálů doporučeno použít IP blok Concat. V dalších částech této práci je tento blok také využit, aby byla demonstrována možnost jeho využití a projevení tohoto připojení v PetaLinux systému.

Po dokončení konfigurace AXI Interrupt Controller je možné aktivovat automatizaci propojení a v konfiguračním okně vybrat požadovanou frekvenci CLK signálů. V této práci je využito 200 MHz. Výsledný design po automatizaci je zobrazen na obr. 12 - 11.



Obr. 12 - 10 Xilinx Vivado – nastavení bloku AXI Interrupt Controller.



Obr. 12 - 11 Xilinx Vivado – block design po automatizaci a propojení bloku AXI Interrupt Controller.

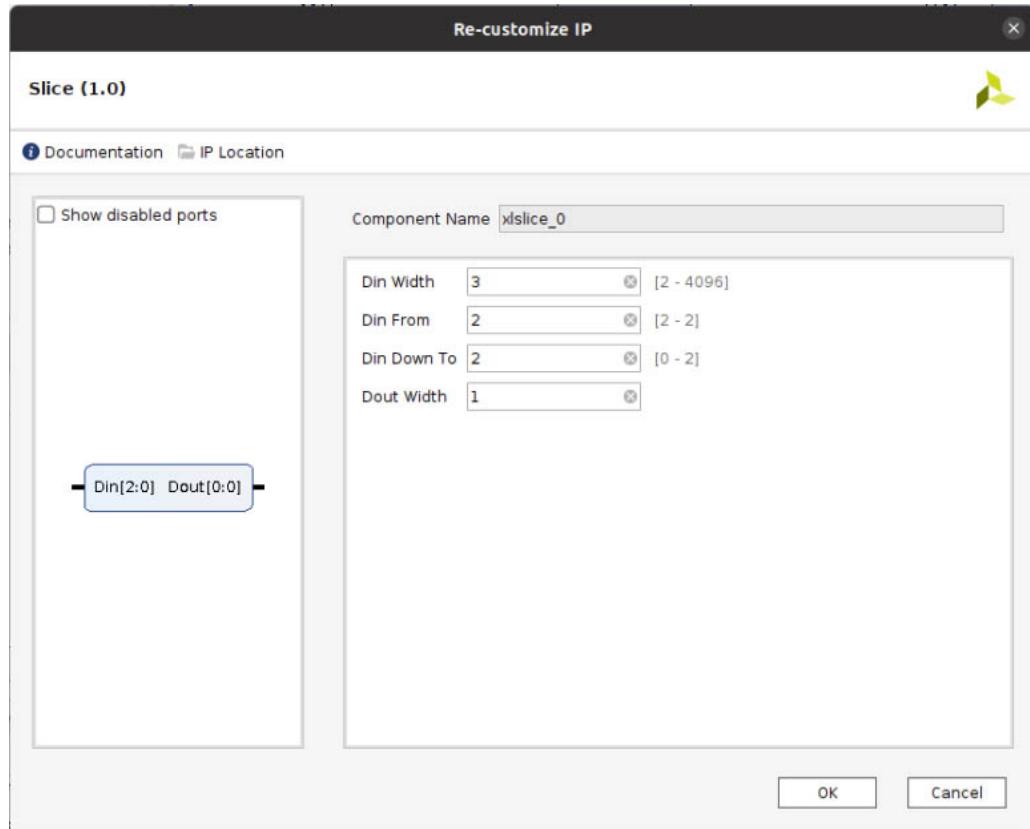
Pokud by vyvájená aplikace nevyužívala další PL IP bloky, je možné přistoupit ke konfiguraci platformy, PS a periferií připojených k PS. Pro vyvájenou ukázkovou aplikaci je vytvořený HW blokový design naznačen v sekci *HW Block design vyvájené aplikace*.

Ovšem při využití vývojové desky Xilinx Kria KR260 je vhodné aktivovat řízení chladícího ventilátoru pomocí PWM. Pro tuto možnost musí být aktivovaný výstup *Waveout* na TTC 0, jak je tomu naznačeno na obr. 12 - 22. Po aktivaci *Waveout* na I/O *EMI0* se objeví na bloku **Zynq UltraScale+ MPSoC** výstup s názvem *emio_ttc0_wave_o[2:0]*. Označení *[2:0]* znamená, že signál je tříbitový. Pro řízení ventilátoru pomocí PWM je ovšem využíván pouze jeden bit. Pro odstranění dvou horních bitů je využito IP bloku **Slice**.

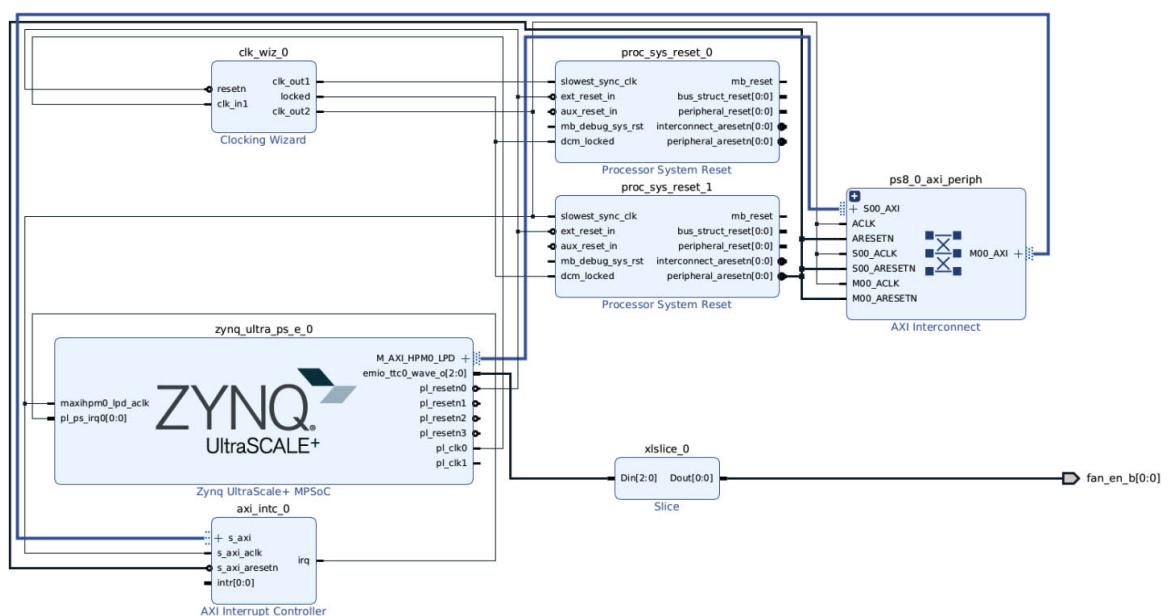
Konfigurace bloku **Slice** je zobrazena na obr. 12 - 12. *Din Width* určuje šířku vstupujícího signálu. V řešeném případě se jedná o tříbitový vstupní signál. *Din From* určuje označení bitu, od kterého bude odstraněno *Din Down To* bitů pro výstup z bloku **Slice**. Vstup *Dout Width* je automaticky doplněn dle nastavení předchozích položek.

Následně je nutné nastavit výstupní pin, na který bude signál PWM pro řízení ventilátoru odesílán. Pro vytvoření pinu existuje mnoho způsobů, v tomto případě je ovšem nejjednodušší pravým tlačítkem myši kliknout na výstup bloku **Slice Dout[0:0]** a zvolit *Make External*. Tento pin je poté vhodné pojmenovat **fan_en_b**. Minimální funkční design s výstupním pinem pro ventilátor je zobrazen na obr. 12 - 13.

Na kartě *Platform Setup* je pro funkční design vhodné aktivovat a pojmenovat základní AXI porty dle tabulky 12 - 1.



Obr. 12 - 12 Xilinx Vivado – nastavení bloku Slice pro odstranění dvou horních bitů signálu.



Obr. 12 - 13 Xilinx Vivado – minimální funkční design s výstupním pinem pro řízení ventilátoru.

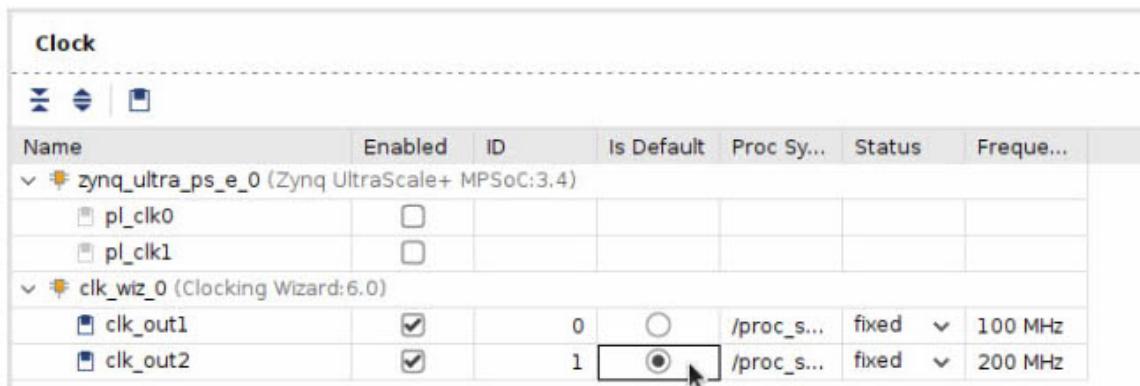
Tab. 12 - 1 Ukázka nastavených AXI portů v Xilinx Vivado platformě pro Xilinx Kria KR260.

Name	Enabled	Mexport	SP Tag
M_AXI_HPM0_FPD	X	M_AXI_GP	-
M_AXI_HPM1_FPD	X	M_AXI_GP	-
S_AXI_HPC0_FPD	X	S_AXI_HPC	HPC0
S_AXI_HPC1_FPD	X	S_AXI_HPC	HPC1
S_AXI_HP0_FPD	X	S_AXI_HP	HP0
S_AXI_HP1_FPD	X	S_AXI_HP	HP1
S_AXI_HP2_FPD	X	S_AXI_HP	HP2
S_AXI_HP3_FPD	X	S_AXI_HP	HP3

Dalším krokem je povolení a nastavení výchozích CLK signálů v záložce *Clock*. V této práci byl zvolen výchozí CLK signál 200 MHz. Snímek záložky zachycující nastavení pro dva clock signály je na obr. 12 - 14.

Posledním důležitým krokem je v kartě *Platform Setup -> Interrupt* povolit signál `intr` z použitého bloku **Axi Interrupt Controller**.

Volitelným, ovšem vhodným krokem je pojmenovat platformu a udat jejího autora a verzi v kartě *Platform Setup -> Interrupt*.



Obr. 12 - 14 Xilinx Vivado – záložka nastavení CLK signálů na platformě.

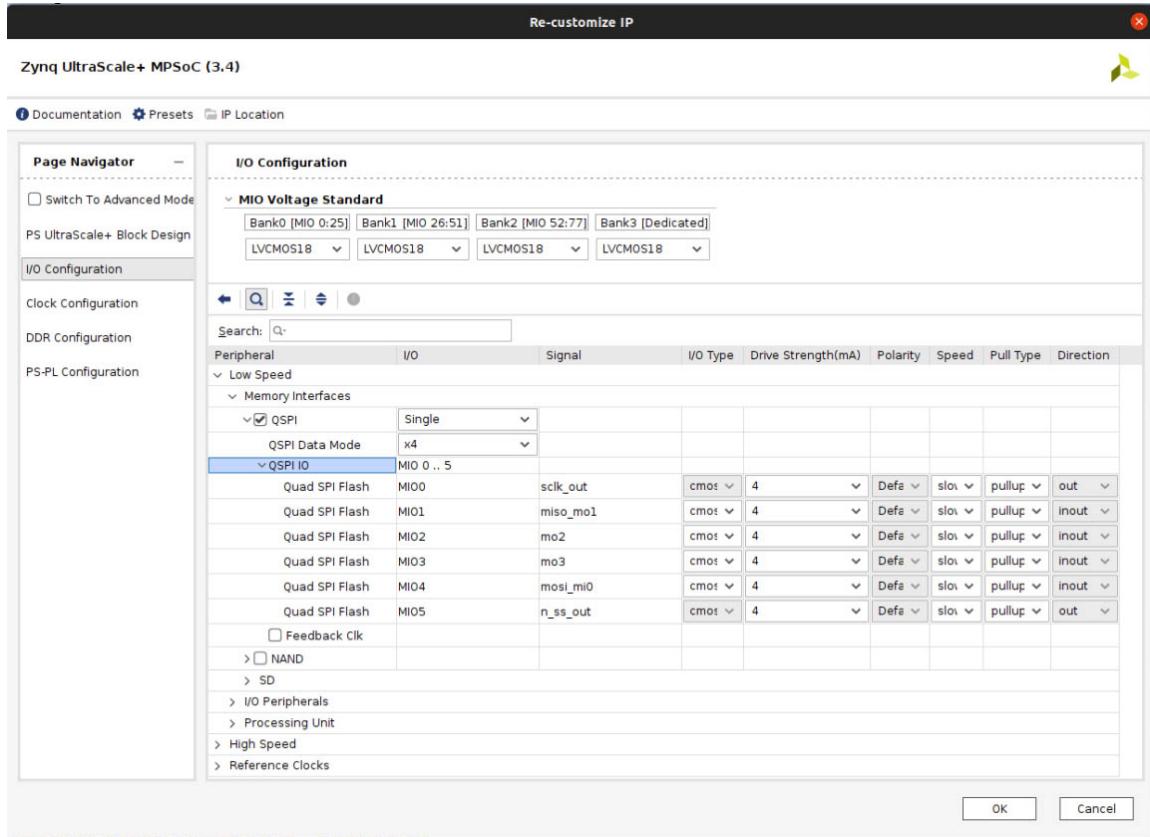
Autorka článku [46] zmiňuje, že pokud je zapnuta možnost *Incremental Synthesis*, můžou vznikat při postupu tvorby akcelerované aplikace problémy. Proto doporučuje pomocí nabídky *Settings -> Synthesis -> Incremental Synthesis* zvolit možnost *Disable incremental synthesis*.

12.2.1 Konfigurace PS pro využití implementovaných periferií

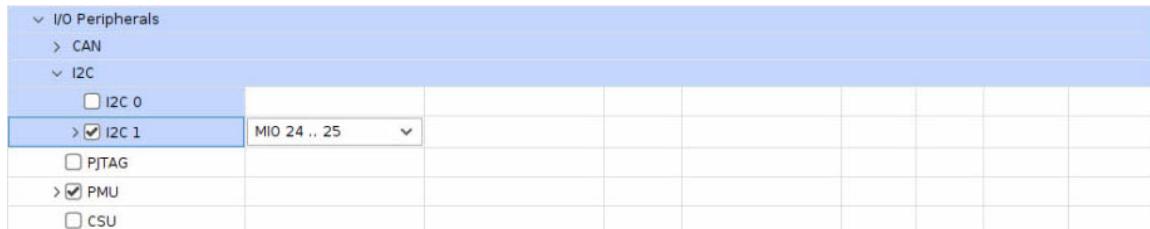
Aby bylo možné využívat periferie, jako je DisplayPort, Ethernet nebo USB konektory, je nutné nakonfigurovat PS. Konfigurační nabídka je otevřena při konfiguraci IP bloku **Zynq UltraScale+ MPSoC** a vybrání *I/O Configuration*.

Pro názornost je využito snímků obrazovky jednotlivých významných konfigurací. Některé z konfigurací jsou již nastaveny jako výchozí, některé je třeba konfigurovat.

Na obr. 12 - 15 je zobrazeno kompletní konfigurační okno ve kterém jsou jednotlivá připojení pro PS nastavovány. Další snímky jsou však z důvodu úspory plochy zaměřeny pouze na konkrétní nastavení.



Obr. 12 - 15 Xilinx Vivado – PS I/O Configuration QSPI zařízení.



Obr. 12 - 16 Xilinx Vivado – PS I/O Configuration I2C zařízení.

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Polarity	Speed	Pull Type	Direction
<input checked="" type="checkbox"/> PMU								
<input type="checkbox"/> GPI EMIO								
<input type="checkbox"/> GPO EMIO								
<input checked="" type="checkbox"/> GPIO 0	MIO 26							
PMU GPIO 0	MIO26	gpi[0]	cmos	12	Defa	fast	pullup	in
<input type="checkbox"/> GPIO 1								
<input type="checkbox"/> GPIO 2								
<input type="checkbox"/> GPIO 3								
<input type="checkbox"/> GPIO 4								
<input checked="" type="checkbox"/> GPIO 5	MIO 31	gpi[5]	cmos	12	Defa	fast	pullup	in
PMU GPIO 5	MIO31	gpi[5]	cmos	12	Defa	fast	pullup	in
<input type="checkbox"/> GPO 0								
<input checked="" type="checkbox"/> GPO 1	MIO 33							
PMU GPO 1	MIO33	gpo[1]	cmos	4	Defa	slow	pullup	out
<input checked="" type="checkbox"/> GPO 2	MIO 34							
<input type="checkbox"/> GPO 3								
Initial State	GP01[3]							
<input type="checkbox"/> GPO 4								

Obr. 12 - 17 Xilinx Vivado – PS I/O Configuration PMU (GPIO 4 a GPIO 5 není vybráno).

SPI									
> <input type="checkbox"/> SPI 0									
> <input checked="" type="checkbox"/> SPI 1	MIO 6 .. 11								
<input checked="" type="checkbox"/> SS[0]	MIO 9								
<input type="checkbox"/> SS[1]									
<input type="checkbox"/> SS[2]									
SPI 1	MIO6	sclk_out	cmos	4	▼	Defa	slo1	pullup	inout
SPI 1	MIO9	n_ss_out[0]	cmos	4	▼	Defa	slo1	pullup	inout
SPI 1	MIO10	miso	cmos	4	▼	Defa	slo1	pullup	inout
SPI 1	MIO11	mosi	cmos	4	▼	Defa	slo1	pullup	inout

Obr. 12 - 18 Xilinx Vivado – PS I/O Configuration SPI zařízení.

UART									
> <input checked="" type="checkbox"/> UART 1	MIO 36 .. 37								
<input type="checkbox"/> MODEM									
UART 1	MIO36	txd	cmos	4	▼	Defa	slo1	pullup	out
UART 1	MIO37	rxd	cmos	12	▼	Defa	fast	pullup	in

Obr. 12 - 19 Xilinx Vivado – PS I/O Configuration UART zařízení.

GPIO									
> <input type="checkbox"/> GPIO EMIO									
> <input checked="" type="checkbox"/> GPIO0 MIO	MIO 0 .. 25								
> <input checked="" type="checkbox"/> GPIO1 MIO	MIO 26 .. 51								
> <input type="checkbox"/> GPIO2 MIO									

Obr. 12 - 20 Xilinx Vivado – PS I/O Configuration GPIO zařízení.

Processing Unit									
> SWDT									
> <input checked="" type="checkbox"/> SWDT 0									
<input type="checkbox"/> Clock in									
<input type="checkbox"/> Reset out									
> <input checked="" type="checkbox"/> SWDT 1									
<input type="checkbox"/> Clock in									
<input type="checkbox"/> Reset out									

Obr. 12 - 21 Xilinx Vivado – PS I/O Configuration System Watchdog Timers (SWDT).

TTC									
> <input checked="" type="checkbox"/> TTC 0									
<input type="checkbox"/> Clock									
<input checked="" type="checkbox"/> Waveout	EMIO								
> <input checked="" type="checkbox"/> TTC 1									
> <input checked="" type="checkbox"/> TTC 2									
> <input checked="" type="checkbox"/> TTC 3									
<input type="checkbox"/> Clock									
<input type="checkbox"/> Waveout									

Obr. 12 - 22 Xilinx Vivado – PS I/O Configuration Triple Timer Counter (TTC), TTC 0 výstup Waveout je využit pro PIN řídící chladicí ventilátor na SOM.

USB	
USB0	
> <input checked="" type="checkbox"/> USB 0	MIO 52 .. 63
> <input checked="" type="checkbox"/> USB 3.0	GT Lane2
USB1	
> <input checked="" type="checkbox"/> USB 1	MIO 64 .. 75
> <input checked="" type="checkbox"/> USB 3.0	GT Lane3
USB Reset	Separate MIO Pin
Reset Polarity	Active Low
> <input checked="" type="checkbox"/> USB 0	MIO 76
> <input checked="" type="checkbox"/> USB 1	MIO 77

Obr. 12 - 23 Xilinx Vivado – PS I/O Configuration USB.

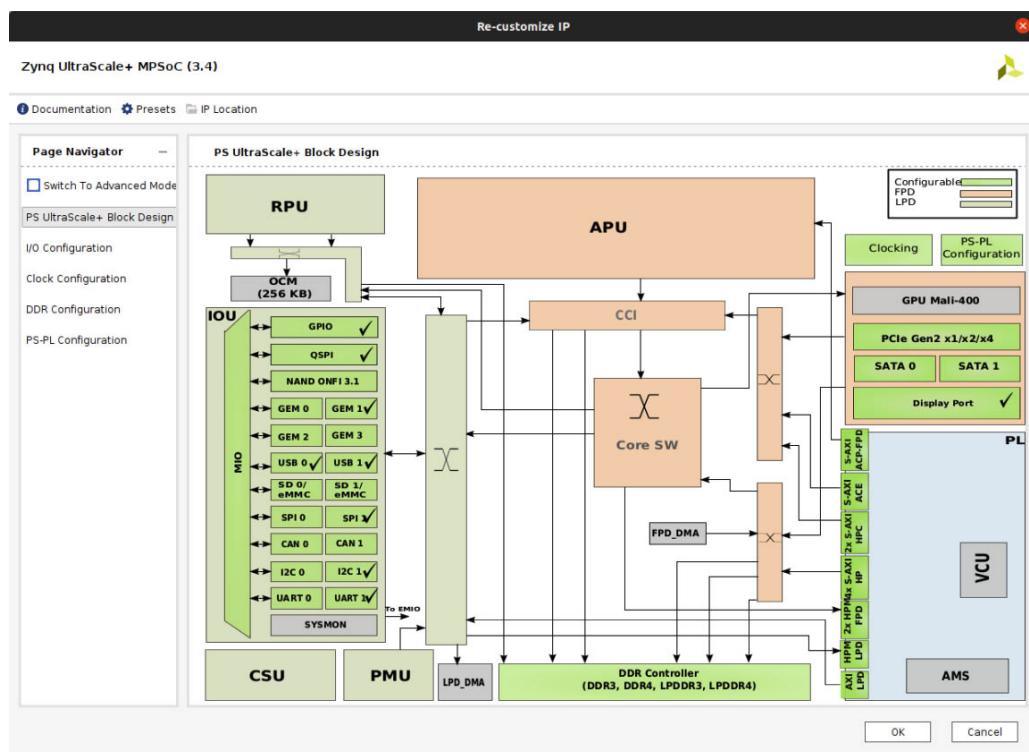
Display Port	
> <input checked="" type="checkbox"/> DPAUX	MIO 27 .. 30
Lane Selection	Single Lower
DP Lane0	GT Lane1

Obr. 12 - 24 Xilinx Vivado – PS I/O Configuration DisplayPort.

Po úspěšné konfiguraci I/O je možné přistoupit ke kartě *PS-PL Configuration* a v záložce *General -> Fabric Reset Enable* povolit *Fabric Reset Enable* a zvolit v nabídce *Number of Fabric Resets* číslo 4.

Nastavení *Number of Fabric Resets* udává, kolik signálů je možné použít pro reset IP bloků realizovaných v PL. [48] [49]

V předchozích krocích byla představena základní konfigurace PS pro funkčnost periferií, přítomných na vývojové desce Xilinx Kria KR260 s Zynq UltraScale+ MPSoC. Po provedení konfigurace je možné v konfigurační nabídce PS IP v kartě *PS UltraScale+ Block Design* pozorovat u konfigurovaných prvků označení pomocí „✓“ symbolu. Ukázka systému je zobrazena na obr. 12 - 25.



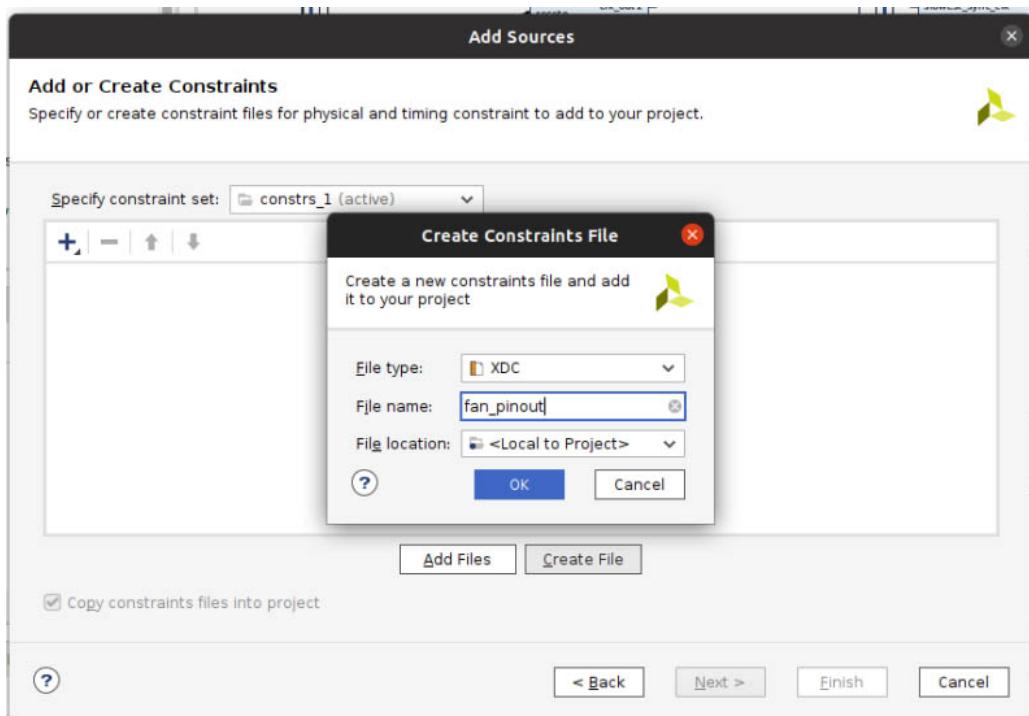
Obr. 12 - 25 Xilinx Vivado – PS UltraScale+ Block Design v PS IP.

12.2.2 Design Constraints

Aby bylo možné vytvořené *virtuální* piny ve Vivado propojit se skutečnými fyzickými piny MPSoC, resp. fyzickými piny vývojové desky, je třeba vytvořit soubor v prostředí Vivado, který toto fyzické propojení definuje. Tyto soubory se nazývají *Constraints Files*.

Jejich tvorba může probíhat i mimo prostředí Vivado, jsou to textové soubory, jež lze upravit v libovolném textovém editoru. Tato skutečnost lze využít při časté tvorbě designů v nových projektech pro zrychlení konfigurace. Soubory je poté možné jednoduše importovat.

Soubory se vkládají pomocí výběru v levém menu *Project Manager -> Add Sources -> Add or create constraints -> Add Files/Create File*. Nabídka tvorby souboru je vyzobrazena na obr. 12 - 26.



Obr. 12 - 26 Xilinx Vivado – okno tvorby/vložení Constraints File.

Pro spojení vytvořeného virtuálního pinu `fan_en_b` a fyzického pinu, ke kterému je na CC připojen ventilátor, je do XDC souboru zapsána konfigurace z kódu 12 - 1.

Kód 12 - 1 byl získán z [45]. Autorem však bylo ověřeno, že se skutečně jedná o fyzický pin A12.

```
1 set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
2
3 # Fan Speed Enable
4 set_property PACKAGE_PIN A12 [get_ports {fan_en_b}]
5 set_property IOSTANDARD LVCMS33 [get_ports {fan_en_b}]
6 set_property SLEW SLOW [get_ports {fan_en_b}]
7 set_property DRIVE 4 [get_ports {fan_en_b}]
```

Kód 12 - 1 Constraints XDC soubor pro přiřazení PL Vivado pinu `fan_en_b` k fyzickému pinu MPSoC na CC.

Vyhledání propojení fyzického pinu a označení pro soubory XDC probíhá totožně pro konektory *PMOD* a *Raspberry Pi HAT*. Oficiální dokumentace pro postup vyhledání pinů nebyla autorem nalezena,

proto se pokusil nalézt vlastní postup, který ověřil na oficiálním fóru Xilinx [50].

Postup nalezení reálného pinu je následující:

1. Nalézt ve schématu [28] Kria KR260 CC požadované označení fyzického pinu (např. pro ventilátor **HDA20**).
2. Nalézt ve schématu [28], ke kterému pinu na PL konektoru (**SOM240_1 CONNECTOR**) je nalezený PIN připojen. (v případě ventilátoru **C24**).
3. V XDC [29] souboru pro Xilinx Kria K26 vyhledat udaný pin v připojení konektoru a zkontrolovat, zda je připojen ke správnému konektoru ze schématu a získat požadované označení **PACKAGE_PIN**.

Po provedení všech pořebných nastavení a konfigurací je možné přejít k finální části postupu ve Vivado.

Nejprve je vhodné vytvořený design validovat pomocí symbolu „✓“ v ovládací liště v okně *Platform*. Pokud jsou získány upozornění s označením *Warning*, je možné pokračovat v tvorbě platformy. V bloku *Sources* zvolit pravým tlačítkem vytvořený block design a aktivovat nabídku *Generate Output Products*. Objeví se konfigurační okno, ve kterém je vhodné nastavit v části *Synthesis Options* volbu *Out of context per IP*. V části *Run Settings* je možné zvolit kolik jader procesoru se bude podílet na zvolené činnosti. Pro osobní počítače autor doporučuje zvolit méně než polovinu dostupných jader CPU. Bylo vyzkoušeno, že pokud jader je zvoleno více, může zatížení systému způsobit samovolné ukončení programu Vivado.

Po ukončení generace je možné zvolit v nabídce pro block design akci *Create HDL Wrapper*. V nabídce akce je výhodné využít možnost *Let Vivado manage wrapper and auto update*, kdy Vivado bude aktualizovat HDL wrapper podle změn provedených v designu.

Autorovi se ovšem při změně block designu vyplatilo HDL Wrapper kompletně smazat a opětovaně provést kroky *Generate Output Products* a *Create HDL Wrapper*.

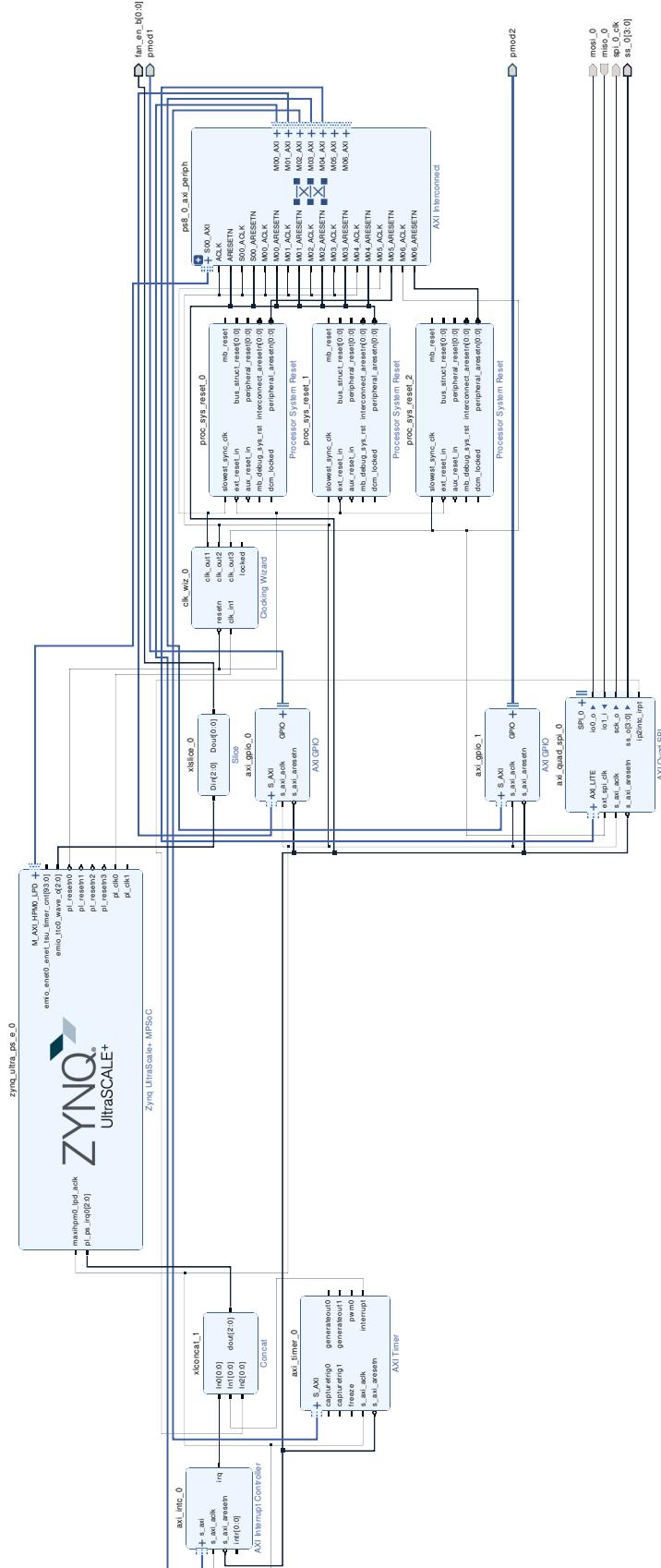
Nyní je již možné přistoupit k syntéze, implementaci a generování bit streamu. Jednotlivé kroky je možné pomocí levé nabídky ve Vivado aktivovat samostatně, nebo zvolit pouze generování bit streamu a Vivado automaticky zajistí, že pokud došlo ke změně designu, budou provedeny kroky syntézy a implementace automaticky.

Po úspěšném provedení generování bitstreamu je možné platformu exportovat pro její další využití v *PetaLinux Tools* a *Vitis*. Export je možné provést pomocí volby *File -> Export -> Export Platform*. Ve zobrazené nabídce je obecně u vývojových desek vhodné zvolit *Hardware and hardware emulation*. Dle [45] podpora HW emulace pro K26 není aktuálně dostupná. Posledním krokem je v nastavení *Platform State* vybrání volby *Pre-synthesis* a *Include bitstream*. Po označení platformy je možné provést export.

V této části byl představen postup tvorby základní HW platfromy pro Xilinx KR260 Starter Kit vývojovou desku v prostředí Vivado. Výstupem projektu ve Vivado je soubor *XPR*, který je dále využit při tvorbě akcelerované aplikace v *PetaLinux Tools* a *Vitis*. Pro některé případy, kdy není potřeba specifický HW v PL je možné využít předpřipravené *XPR* soubory. Tyto předpřipravené soubory slouží pouze k seznámení s vývojovými deskami a nejsou dostačující pro vývoj specifické akcelerované aplikace.

12.2.3 HW block design vyvíjené aplikace

Představený design na obr. 12 - 27 je využit pro tvorbu platformy pro akcelerované aplikace v této práci. Platforma využívá SPI komunikaci, realizovanou v PL a blok AXI Timer.



Obr. 12 - 27 Xilinx Vivado – HW block design vyvíjené aplikace.

13 Tvorba PetaLinux

Jak již bylo zmíněno v části *Aplikace a operační systém*, aplikace pro Xilinx Kria je možné vytvářen pro *Bare Metal / Standalone*, *PetaLinux* a také distribuci operačního systému *Linux Ubuntu*. V této práci je využito systému *PetaLinux*, který díky své tvorbě pomocí *PetaLinux Tools* je možné konfigurovat tak, aby využíval HW konfigurovaný v PL a splňoval požadavky vytvářené aplikace.

Konfigurace pro jednotlivé požadavky vytvářených aplikací se mohou odlišovat, ovšem rámec (flow) tvorby systému zůstává pro danou platformu zachován. V této práci bude představen postup tvorby *PetaLinux* systému pro HW platformu obsahující prvky, představené v části *HW block design vyvíjené aplikace*.

Postup tvorby *PetaLinux* systému čerpá informace z [44], [51] a z experimentálních zjištění autora práce.

Prvním krokem je aktivace *PetaLinux* prostředí (environment). Příkaz k aktivaci je závislý na umístění instalovaných *PetaLinux Tools* a je zobrazen v kódu 13 - 1. Představený postup předpokládá práci s definovanou strukturou projektu, definovanou v části *Struktura složek* v kódu 11 - 1.

```
1 source /tools/Xilinx/PetaLinux/2022.2/settings.sh
```

Kód 13 - 1 Aktivace prostředí PetaLinux verze 2022.2.

Po aktivaci prostředí je možné vytvořit projekt pomocí příkazu 13 - 2. Soubor **xilinx-kr260-starterkit-v2022.2-10141622.bsp** (BSP) je pro aktuální využívanou verzi vývojových nástrojů možné stáhnout z oficiálních stránek Xilinx [35].

```
1 # general command
2 petalinux-create --type project -s <path-to-bsp-file> --name <project-name>
3
4 # example command for defined file structure
5 petalinux-create --type project -s ./../xilinx-kr260-starterkit-v2022
   .2-10141622.bsp --name petalinux
```

Kód 13 - 2 Tvorba PetaLinux projektu ze základního BSP souboru.

13.1 Hardware konfigurace PetaLinux

Nyní je možné přejít do adresáře projektu pomocí `cd <project-name>` a začít konfigurovat *PetaLinux*.

První konfigurace spočívá v načtení HW platformy do projektu pomocí příkazu 13 - 3. Kdy je předpokládáno, že ve složce hw je umístěn soubor exportované platformy z Vivado, získaný v části *Tvorba HW designu pro Xilinx Kria KR260 vývojovou desku*.

```
1 petalinux-config --get-hw-description=../hw/
```

Kód 13 - 3 Konfigurace PetaLinux pomocí XPR souboru z Vivado.

Po načtení XSA souboru je v terminálu zobrazena konfigurační nabídka. Nastavení prvků v této nabídce je pro K26 SOM uvedeno v kódu 13 - 4.

```
1 FPGA Manager -> Fpga Manager <*>
2
3 Image Packaging Configuration -> Root Filesystem Type --> INITRD <*>
```

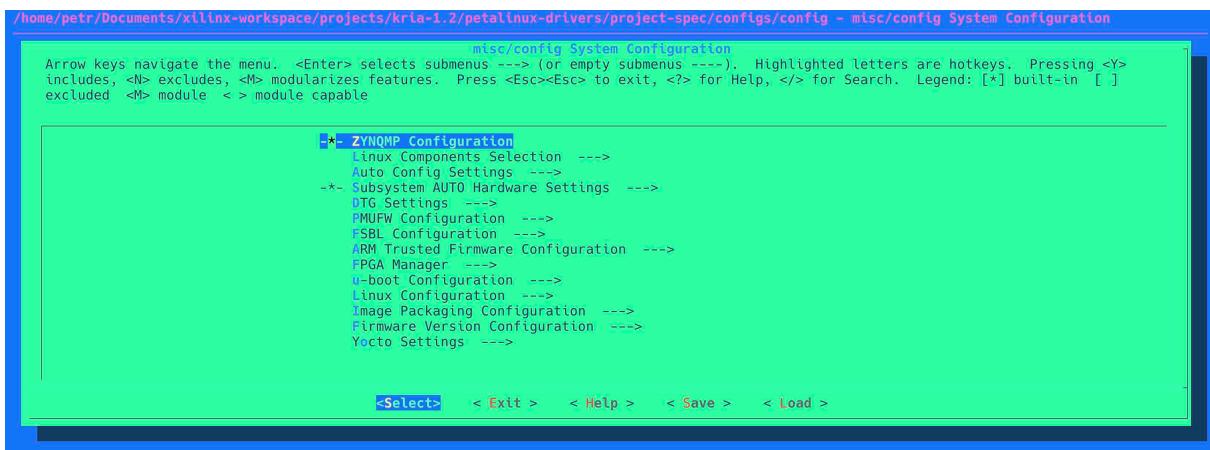
```

4 Image Packaging Configuration -> INITRAMFS/INITRD Image name -> petalinux-
    initramfs-image <*>
5 Image Packaging Configuration -> Copy final images to tftpboot < >
```

Kód 13 - 4 Nastavení v *petalinux-config* pro Xilinx K26 SOM.

Pátý řádek popisuje vypnutí možnosti kopírování souborů pomocí **tftpboot**. Tato možnost nebyla v této práci využita a prozkoumána.

Ukázka konfigurační nabídky, která je zobrazena uživateli je na obr. 13 - 1.



Obr. 13 - 1 Xilinx Vivado – snímek obrazovky konfigurace PetaLinux pomocí „*petalinux-config*“ příkazu.

Bylo zjištěno, že při tvorbě systému s aplikovaným RT patchem pro *PetaLinux* je vhodné provést úvodní build systému bez dodatečných konfigurací a poté build opakovat s požadovanými konfiguracemi. Build proces je spuštěn příkazem 13 - 5.

Proces aplikace RT patche je představen v části *Postup aplikace PREEMPT_RT patch*.

```
1 petalinux-build
```

Kód 13 - 5 PetaLinux build příkaz pro vytvoření systému.

13.2 Konfigurace jádra PetaLinux

Po aplikaci patche je vhodné pokračovat nastavením jádra (pro jádro systému je zavedeno také označení *kernel*) *PetaLinux* systému pomocí příkazu 13 - 6. Ve vytvářené aplikaci je využíván **Userspace IO Driver** a proto je nutné aktivovat konfiguraci jádra, která zajistí automatické načtení driveru (ovladače) do systému při jeho startu. Pokud by se tak nestalo, bylo by nutné vkládat moduly manuálně pomocí příkazů **modprobe** či **insmod**. Při konfiguraci jádra je možné aktivovat i další prvky, které nejsou v této práci využívány. Ukázka konfigurace pro **Userspace IO Driver** je zobrazena v kódu 13 - 7.

```
1 petalinux-config -c kernel
```

Kód 13 - 6 PetaLinux příkaz pro konfiguraci jádra systému.

```

1 Device drivers -> user space IO drivers <\*>
2 Device drivers -> user space IO drivers -> Userspace platform driver with
    generic irq and dynamic memory <*>
3 Device drivers -> user space IO drivers -> Userspace platform driver with
    generic IRQ handling <*>
```

Kód 13 - 7 PetaLinux konfigurace pro User Space IO Driver.

13.2.1 Konfigurace Device Tree

Aby bylo možné využívat zmínovaný **Userspace IO Driver** v podobě **generic-uio** driveru pro řešení přerušení (interrupts) od prvků AXI Quad SPI [52], AXI Timer [53] a dalších, je nezbytné provést odpovídající konfiguraci **devicetree**. Konfiguraci **devicetree** je možné provádět i u tvorby Linux systému pro jiné vývojové desky s SoC, které neobsahují FPGA, ale využívají systému Linux.

Problematika **devicetree** je značně obsáhlá, tento standard je obhospodařován komunitou systému Linux a pravidelně jsou vydávány specifikace, které tuto datovou strukturu popisují. Tyto specifikace je možné nalézt na oficiálních stránkách této komunity. [54]

Existuje i záznam velmi přínosné prezentace z *Embedded Linux Conference Europe* ohledně problematiky Device Tree od *Thomas Petazzoni* s názvem *Device Tree for Dummies!* [55].

Při tvorbě akcelerované aplikace v této práci byl využit DT soubor automaticky vytvořený při konfiguraci jádra systému při prvním build procesu *PetaLinux* systému. Ovšem aby bylo možné použít zmínované bloky, je třeba do souboru **system-user.dtsi** zapsat požadované konfigurace. Soubor **system-user.dtsi** překonfiguruje automaticky vytvořený soubor **system-conf.dtsi**. V hierarchii načítání konfiguračních souborů je čten **system-conf.dtsi** jako poslední.

Upravený obsah **system-user.dtsi** souboru je zobrazen v 13 - 8. Soubor je možné nalézt v cestě `<petalinux-project>/project-spec/meta-user/recipes-bsp/device-tree/files/system.user.dtsi`.

```
1 /include/ "system-conf.dtsi"
2 {
3     chosen {
4         bootargs = "earlycon clk_ignore_unused uio_pdrv_genirq.of_id=
5             generic-uio";
6         stdout-path = "serial0:115200n8";
7     };
8
9     timer@0080020000 {
10         compatible = "axi_timer_0, generic-uio, ui_pdrv";
11         status = "okay";
12     };
13
14     axi_quad_spi@80040000 {
15         compatible = "axi_quad_spi_0, generic-uio, ui_pdrv";
16         status = "okay";
17     };
}
```

Kód 13 - 8 Obsah souboru **system-user.dtsi**, překonfigurovávající soubor **system-conf.dtsi**.

Tento rekonfigurační DT byl získán reengineeringem souboru **pl.dtsi**. Soubor **pl.dtsi** je získán za pomoci exportovaného *XPR* souboru pomocí **XSCT** (Xilinx Software Command-Line Tool). Postup tvorby byl získán z [51] a [44].

V definované struktuře z části *Struktura složek* je vhodné příkazy vyvolávat ze složky **linux-files**. Postup spočívá ve spuštění nástroje, načtení HW designu ze souboru *XSA* a použití příkazu **createdts** s příslušnými parametry. Postup je naznačen v kódu 13 - 9.

```
1 xsct # start the Xilinx Software Command-Line Tool
```

```

2
3 hsi::open_hw_design ../hw/kria_bd_wrapper.xsa # open HW design
4
5 createdts -hw ../hw/kria_bd_wrapper.xsa -zocl -platform-name kria_kr260
   -git-branch xlnx_rel_v2022.2 -overlay -compile -out ./dtg_output #
   create devicetree overlay from defined hardware with the help of Xilinx
   official GitHub repository
6
7 exit # exit the tool

```

Kód 13 - 9 Postup tvorby pl.dtsi souboru pomocí XSCT, popisující DT při běžícím PetaLinux.

Uvedeným způsobem je získán textový soubor `pl.dtsi`, který je možné opět konfigurovat pomocí textového editoru. Soubor má podobnou strukturu jako běžný DT, ovšem definuje tzv. fragmenty pro `device tree overlay`. Protože byla v kroku HW konfigurace *PetaLinux* aktivována možnost `FPGA Manager -> Fpga Manager <*>`, je v jádru systému *PetaLinux* zabudovaná jen základní část DT. Zbytková část je poté načtena do systému při běhu systému (at runtime) po vyvolání příkazu `xmutil loadapp <name-of-the-app>` a spuštění aplikace, resp. načtení bitstreamu do PL. Příkaz `xmutil` je specifický pro *PetaLinux* a Xilinx nástroje. Pro ostatní *Linux* distribuce a zařízení je nutné využívat obecný způsob načítání `device tree overlay`.

Pro vyvíjenou aplikaci v této práci je v `pl.dtsi` souboru nutné u použitého bloku AXI Timer změnit *property value* na `compatible = "generic-uio"`; a pro AXI Quad SPI na téže hodnotu.

Pro funkčnost v *PetaLinux* je nutné textový soubor `pl.dtsi` převést do binární podoby. Ke komplikaci je používán nástroj `dtc` (Device Tree Compiler), dostupný pro většinu distribucí *Linux* systému. Příkaz, který popisuje komplikaci souboru `pl.dtsi` (Device Tree Source Include, dtsi) na `pl.dtbo` (Device Tree Blob Object, dtbo) je zobrazen v kódu 13 - 10.

```

1 dtc -@ -O dtb -o <path-to-output-file> <path-to-input-file> # general
   command
2
3 dtc -@ -O dtb -o ./dtg_output/dtg_output/kria_kr260/psu_cortexa53_0/
   device_tree_domain/bsp/pl.dtbo ./dtg_output/dtg_output/kria_kr260/
   psu_cortexa53_0/device_tree_domain/bsp/pl.dtsi # command for project
   file structure

```

Kód 13 - 10 Kompilace textového souboru device tree overlay pl.dtsi na binární pl.dtbo soubor.

Pokud byla komplikace úspěšná (v některých případech dochází k vypsání hlášky ohledně IP bloku `interrupt controller`, tu je možné ignorovat) je vhodné přesunout soubor `pl.dtbo` do složky `transfer`.

13.3 Konfigurace Root File System

Po úspěšné konfiguraci kernelu *PetaLinux* s aktualizovaným souborem `system-user.dtsi` je možné přistoupit ke kroku konfigurace *root filesystem* (`rootfs`). Konfigurační nabídka je vyvolána pomocí příkazu 13 - 11.

```

1 petalinux-config -c rootfs

```

Kód 13 - 11 Příkaz pro vyvolání konfigurace root filesystem

Při konfiguraci `rootfs` je možné vybrat, jaké aplikace budou do systému *PetaLinux* v základní instalaci vloženy. Pro funkční akcelerovanou aplikaci je doporučováno zvolit výběr nastavení, uvedený v kódu 13 - 12.

Pokud vývoj aplikace nevyžaduje nejnovější instalační balíky a jejich aktualizace při připojení k internetu, není třeba instalovat `dnf` (Dandified YUM – new version of Yellowdog Updater, Modifier, Package Manager).

Základní funkce jednotlivých nastavení je následující:

- `dnf` – Package Manager,
- `xrt` – Xilinx Runtime Library – knihovna zajišťující komunikaci mezi PS aplikací a akceleroványmi kernely v PL,
- `zocl` – ovladač pro ZynQ OpenCL akcelerátory,
- `opencl-headers` a `opencl-clhpp` – knihovny pro OpenCL (Open Computing Language),
- `packagegroup-petalinux` – skupina balíků pro *PetaLinux*,
- `packagegroup-petalinux-opencv` – skupina balíků pro OpenCV knihovnu, zajišťující real-time optimalizované zpracování obrazu (není nutné pro vyvýjenou aplikaci),
- `packagegroup-petalinux-v4lutils` – skupina balíků, zajišťující práci mediálních zařízení jako jsou webkamery apod. (není nutné pro vyvýjenou aplikaci),
- `packagegroup-petalinux-x11` – skupina balíků, zajišťující X Window System (není nutné pro vyvýjenou aplikaci).

Aktivaci automatického přihlašování uživatele `root` není doporučeno z bezpečnostních důvodů používat u zařízení v provozu. Při debugingu a vyvíjení aplikace ovšem automatické přihlášení přináší zjednodušení. Bezpečnost systému v produkčním prostředí není obsahem této práce.

```
1 Image Features -> auto-login <*> # do not use in a production
2
3 Filesystem Packages -> base -> dnf -> dnf <*>
4
5 Filesystem Packages -> x11 -> libdrm -> libdrm <*>
6
7 Filesystem Packages -> x11 -> libdrm -> libdrm-tests <*>
8
9 Filesystem Packages -> x11 -> libdrm -> libdrm-kms <*>
10
11
12 Filesystem Packages -> libs -> xrt -> xrt <*>
13
14 Filesystem Packages -> libs -> xrt -> xrt-dev <*>
15
16 Filesystem Packages -> libs -> zocl -> zocl <*>
17
18 Filesystem Packages -> libs -> opencl-headers -> opencl-headers <*>
19
20 Filesystem Packages -> libs -> opencl-clhpp -> opencl-clhpp-dev <*>
21
```

```

22
23 Petaliunx Package Groups -> packagegroup-petalinux -> <*>
    packagegroup-petalinux
24
25 Petaliunx Package Groups -> packagegroup-petalinux-opencv ->
    packagegroup-petalinux-opencv <*>
26
27 Petaliunx Package Groups -> packagegroup-petalinux-v4lutils ->
    packagegroup-petalinux-v4lutils <*>
28
29 Petaliunx Package Groups -> packagegroup-petalinux-x11 ->
    packagegroup-petalinux-x11 <*>

```

Kód 13 - 12 Doporučené nastavení rootfs pro akcelerovanou aplikaci.

13.4 Závěrečný build PetaLinux, generování SDK a tvoření WIC obrazu systému

Po provedení požadované `rootfs` konfigurace je možné přistoupit opět k build procesu pomocí příkazu 13 - 5. Po první části build procesu je možné spustit proces generování *SDK* (Software Development Kit), který je nutný k tomu, aby bylo možné akcelerovanou aplikaci ve Vitis vyvíjet. Generování *SDK* je provedeno pomocí příkazu 13 - 13. Autor vyzdvíhal, že pokud je postup tvorby *PetaLinux* dodržen a jsou nainstalované všechny správné packages v operačním systému, kde dochází k tvorbě *SDK*, a vznikne v procesu generování chyba v `do_populate_sdk` zahrnující klíčové slovo `python`, je třeba provést restart operačního systému a pokusit se o opakování příkazu 13 - 13. Příčina této chyby nebyla odhalena.

Po build procesu *SDK* je pro tvorbu aplikace ve Vitis IDE nutné *SDK* také instalovat. Je doporučeno provést instalaci do složky `linux-files` pomocí příkazu 13 - 14. Po instalaci *SDK* je možné přistoupit ke kroku vytvoření a „zabalení“ (package) určitých komponent, které slouží k vytvoření obrazu systému *PetaLinux*. K tomuto kroku jsou využity příkazy 13 - 15 a 13 - 16.

```
1 petalinux-build --sdk
```

Kód 13 - 13 Příkaz pro aktivování build procesu SDK

```
1 ./sdk.sh -d ../../../../../../linux-files/
```

Kód 13 - 14 Příkaz pro instalaci SDK

```
1 petalinux-package --boot --u-boot --force
```

Kód 13 - 15 Příkaz pro zabalení boot komponent pro tvorbu obrazu systému.

```
1 petalinux-package --wic --images-dir ./images/linux/ --bootfiles "ramdisk.
    cpio.gz.u-boot,boot.scr,Image,system.dtb,system-zynqmp-sck-kr-g-revB.dtb
    " --disk-name "sda"
```

Kód 13 - 16 Příkaz pro vytvoření obrazu systému, který bude využit v procesu flash SD Card (vybalování obrazu systému na SD kartu).

Aby bylo možné využít vygenerované SDK pomocí Vitis IDE, je nutné vytvořené boot komponenty projektu překopírovat do jedné složky, aby při tvorbě Vitis Platformy mohly být jejich cesty snadno vloženy do konfiguračního souboru. Boot komponenty jsou umístěny v cestě `<petalinux-project>/images/linux` a je vhodné je zkopirovat do složky ve vytvořené struktuře dle příkazu 13 - 17.

```
1 cp bl31.elf pmufw.elf system.dtb u-boot.elf zynqmp_fsbl.elf ../../../../
   linux-files/pfm/boot/
```

Kód 13 - 17 Příkaz pro kopírování boot komponent do složky dané strukturou projektu.

Vygenerovaný obraz systému typu **WIC** je možné použít na flash SD karty, která je poté vložena do slotu vývojové desky. K flashnutí je možné na operačním systému macOS nebo Linux využít příkazu `dd`. Pokud uživatel není zkušený, je doporučováno využít program *balenaEtcher*.

V této části byl představen postup generování systému *PetaLinux*, byly doporučeny jednotlivá nastavení a konfigurace, vhodné pro vyvýjenou aplikaci. Na závěr části byl nastíněn postup flash *PetaLinux* systému na SD kartu, kterou je po dokončení flash procesu možné vložit do vývojové desky a započít boot proces *PetaLinux* systému.

13.5 Spuštění systému PetaLinux na KR260

Po úspěšném procesu flashnutí obrazu systému **WIC** na SD kartu, je možné kartu vložit do příslušného slotu vývojové desky a připojit desku na napájení. CC KR260 neobsahuje přepínač, který by umožňoval přerušení napájení desky. Proto se deska vypíná pomocí manuálního odpojení napájení. Restart (reboot) desky je proveden softwarovým způsobem pomocí příkazu `reboot now` nebo opět pomocí odpojení a připojení napájení (power cycle).

Po připojení napájení dochází k automatickému spuštění *bootloaderu* a postupnému načtení a spuštění systému. Pokud je uživatel připojen k desce pomocí USB, resp. UART komunikace (např. pomocí *minicom* s baud rate 115200), budou v terminálu vypisovány informace z bootloaderu před spuštěním *PetaLinux* i informace z *kernel ring buffer*. Po úspěšném spuštění systému je možné informace z *kernel ring buffer* opět zobrazit pomocí příkazu `dmesg`. Výpis je v některých případech důležitý při debugingu systému a jeho informace je možné využít i při spuštění akcelerované aplikace a načítání `pl.dtbo` a `xclbin` souboru.

13.6 Nastavení Ethernet adaptéru po spuštění systému

Bыло vyzváno, že vlivem tvorby platformy a nastavení *PetaLinux* je `eth0` interface po restartu systému neaktivní a pro funkční komunikaci je třeba jej aktivovat pomocí příkazu `ifup eth0`.

Pokud v síti, ke které je připojena vývojová deska není aktivní DHCP server, je nutné upravit soubor v cestě `/etc/network/interfaces` a vložit požadované nastavení adaptéra. Ukázka nastavení pro vývojové pracoviště autora je v kódu 13 - 18. Popis architektury pracoviště je popsán v části *Popis pracoviště*.

```
1 auto eth0
2 iface eth0 inet static
3   address 192.168.144.165
4   netmask 255.255.255.0
5   network 192.168.144.0
6   gateway 192.168.144.1
```

7 broadcast 192.168.144.255

Kód 13 - 18 Nastavení eth0 interface pro KR260 vývojovou desku na vývojovém pracovišti autora.

Autor práce však požadoval automatickou inicializaci adaptéru eth0 a proto využil možnost vytvořit shell skript, který bude spouštěn po startu *PetaLinux* systému. Jedná se ovšem o experimentální přístup, protože po některých restartech systému vznikla situace, kdy skript byl spuštěn, adaptér byl načtený (kontrola přiřazené IP adresy proběhla pomocí ip addr), ale k zařízení nebylo možné se připojit. Důvod tohoto problému nebyl při realizaci této práce objeven.

Obsah vytvořeného skriptu **ethernetUp.sh** je zobrazen v 13 - 19. Soubor **networking** je již zakomponován v systému a obsahuje potřebné konfigurace.

```
1 #!/bin/sh
2 echo "Starting network interfaces!"
3 ip addr
4 /etc/init.d/networking start
5 ip addr
```

Kód 13 - 19 Skript pro automatickou inicializaci adaptéru eth0.

Příkazy, které je třeba vyvolat, aby skript **ethernetUp.sh** byl spuštěn po restartu systému, jsou uvedeny v kódu 13 - 20.

```
1 sudo cp /path/to/your-script/ethernetUp.sh /etc/init.d/ # copying script to
   init.d directory
2
3 sudo chmod +x /etc/init.d/ethernetUp.sh # making script executable
4
5 sudo ln -s /etc/init.d/ethernetUp.sh /etc/rc5.d/S99ethernetUp.sh # creating
   soft symbolic link to a script in the folder rc5.d which represents
   layer, when the script should be run
```

Kód 13 - 20 Skript pro automatickou inicializaci adaptéru eth0.

14 Tvorba akcelerované aplikace ve Vitis IDE

V předchozích kapitolách bylo představeno, jakým způsobem připravit HW platformu a *PetaLinux* systém pro vývoj akcelerované aplikace. V této části je uveden postup vytvoření akcelerované aplikace pomocí Vitis, s využitím HLS.

HLS může být použito také v prostředí Vitis HLS, ve kterém je možné kromě **xo** souborů pro **v++ linker** vytvářet RTL IP PL bloky do prostředí Vivado. Uvedené možnosti nebylo v této práci využito a bude předmětem možných navazujících prací.

Pro vytvoření **c++** aplikace pro PS a kernelu pro PL, který je pomocí HLS z **c++** zkompilován do objektu **xo**, je v této práci využito prostředí Vitis IDE. Autorem práce je doporučováno využít IDE v maximální míře. Při tvorbě aplikace a prozkoumávání možností využití platformy autor narazil na problémy s odezvou od GUI Vitis IDE, proto přešel částečně na headless řešení pomocí příkazové řádky. Headless řešení umožňuje rychlejší postup při vytváření aplikace, který je možné automatizovat pomocí **bash** skriptů. Příkazová řádka v této práci byla převážně využívána ke kompliaci souborů pro PS, kernel a k procesu linkování jednotlivých artefaktů pomocí **v++ linkeru**.

Aby bylo možné Vitis IDE spustit v operačním systému Linux, je nutné aktivovat prostředí pomocí příkazu 14 - 1. Poté je možné spustit Vitis IDE pomocí příkazu **vitis**. Po spuštění je třeba zvolit *Workspace*, které je doporučeno volit do složky **vitis** definované ve *Struktura složek*.

14.1 Tvorba platformy pro akcelerovanou aplikaci

Před tvorbou samotné akcelerované aplikace je nejprve nutné vytvořit tzv. *Platform project*, jež využije vytvořený Vivado *XSA* soubor

a soubor *Image* z cesty <*petalinux-project*>/linux/image/*Image*. Při tvorbě platformy v nabídce výběru systému je nutné vybrat *linux* a zrušit možnost *Generate boot components*. Konfigurace je zobrazena na obr. 14 - 1.

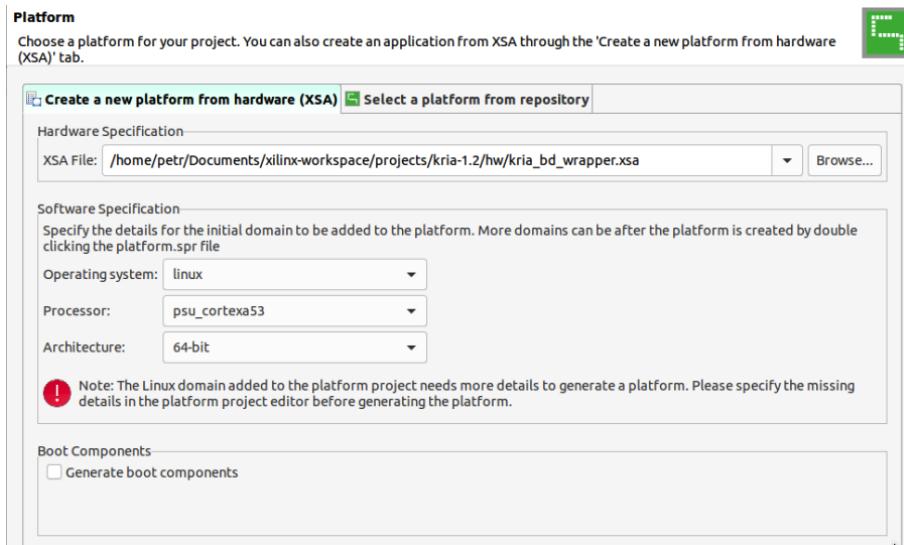
Po vytvoření *Platform project* je nezbytné do projektu vložit cesty potřebných souborů platformy. Přidání cest probíhá v souboru <*platform-name*>/platform.spr v záložce **linux on cortexa53**. Vysvětlení, jaké cesty souborů je třeba vložit v konfiguraci, je uvedeno v tabulce 14 - 1.

```
source /tools/Xilinx/Vitis/2022.2/settings64.sh
```

Kód 14 - 1 Aktivace prostředí pro Vitis verze 2022.2.

Tab. 14 - 1 Nastavení cest souborů pro platformu ve Vitis IDE v souboru *platform.spr*.

Jiméno	Cesta	Pozmáka
OS	linux	generované automaticky
Processor	psu_cortexa53	generované automaticky
Supported Runtimes	OpenCL	generované automaticky
Display Name	linux on psu_cortexa53	generované automaticky
Description	linux_domain	generované automaticky
BiF File	-	generovat pomocí šipky tlačítka
Boot components directory	<project-name>/linux-files/pfm/boot	-
Linux Rootfs	<project-name>/<petalinux-project>/images/linux/rootfs.ext4	generované pomocí <i>SDK</i>
Bootmode	SD	generované automaticky
FAT32 Partition Directory	<project-name>/linux-files/pfm/sd_card	složka zůstane prázdná
Sysroot Directory	<project-name>/linux-files/sysroots/cortexa72-cortexa53-xilinx-linux/	generované pomocí <i>SDK</i>
QEMU Data	-	generované automaticky
QEMU Arguments	-	generované automaticky
PMU QEMU Arguments	-	generované automaticky

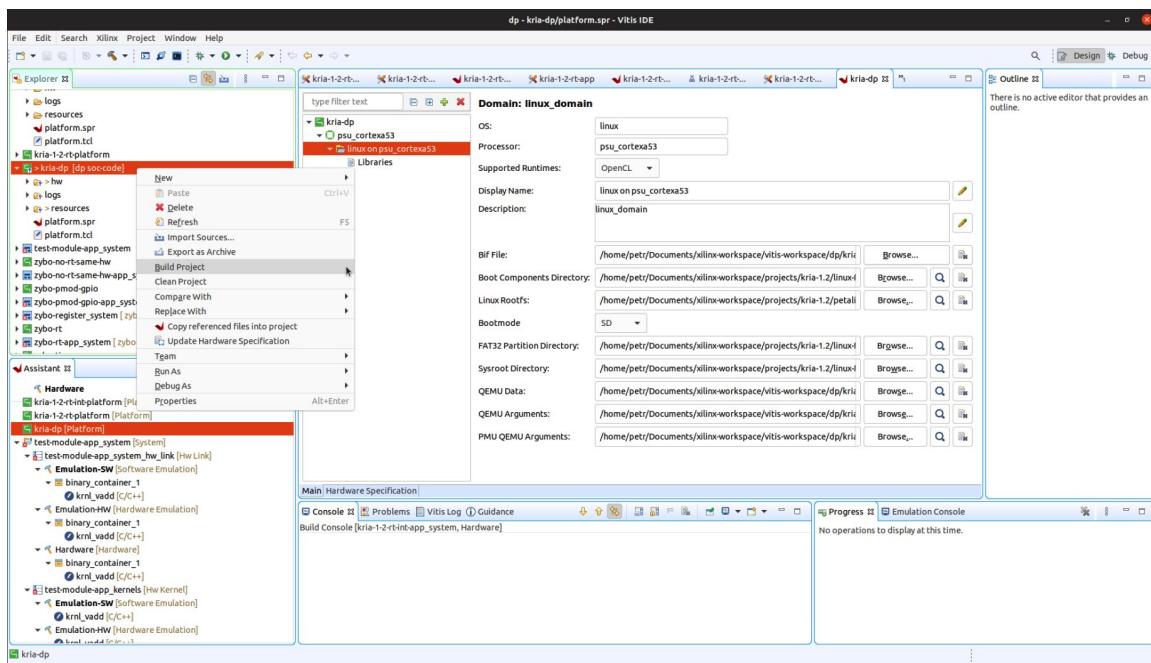


Obr. 14 - 1 Xilinx Vitis IDE – tvorba platformy pro akcelerovanou aplikaci.

Většinu souborů tvořených ve Vitis je možné otevřít pomocí textového editoru.

Soubor **platform.spr** není výjimkou. Pokud dochází k tvorbě mnoha projektů se stejným nastavením platformy, je možné vytvořit skripty, které automaticky vytvoří konfigurovaný **platform.spr** soubor s danou strukturou a nastavením.

Po základní konfiguraci platformy je možné spustit její kompliaci pomocí pravého kliknutí na její název v okně *Explorer* a volby možnosti *Build Project*. Ukázka tohoto kroku je na obr. 14 - 2.



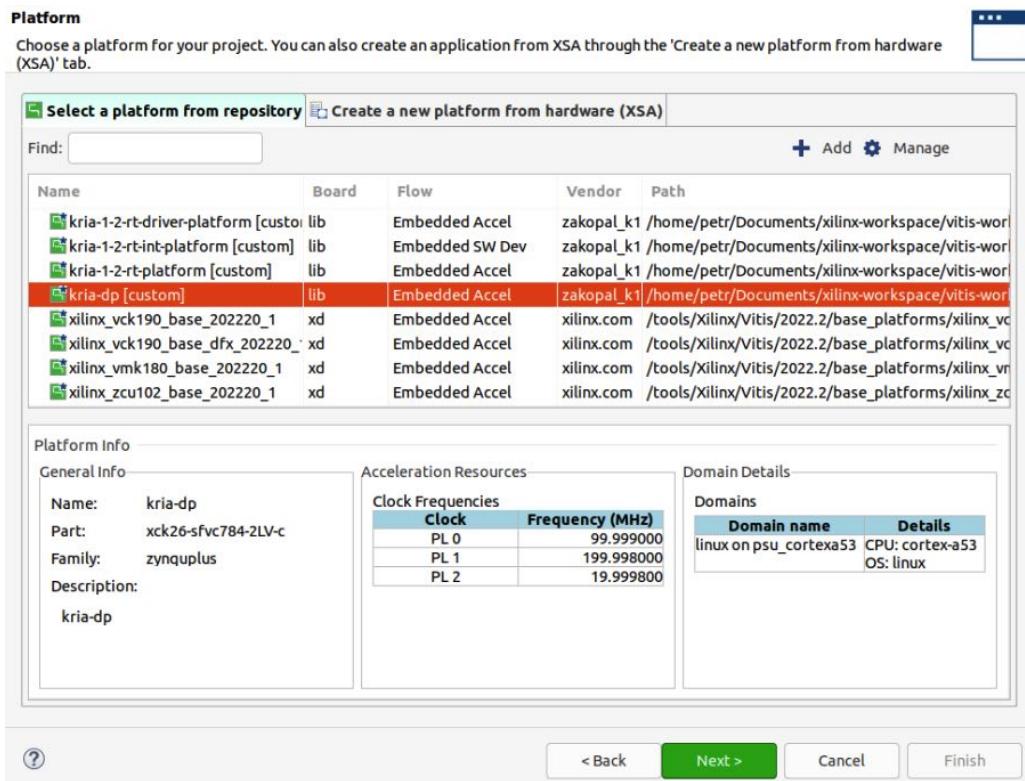
Obr. 14 - 2 Xilinx Vitis IDE – build proces platfromy.

14.2 Tvorba application project

Po dokončení build procesu platformy je již možné přejít k tvorbě *Application Project*. *Application Project* je možné vytvářet pouze na předem vytvořenou a zkompilovanou platformu. Ukázka výběru platformy

s povolenou HW akcelerací, na kterou může být vytvářen *Application Project*, je na obr. 14 - 3.

Při inicializaci projektu v kroku *Domain* je nutné zvolit v části *Application settings* cestu k položce *Kernel Image*, ta je pro vyvíjenou aplikaci umístěna v cestě
`<project-name>/<petalinux-project>/images/linux/Image`.



Obr. 14 - 3 Xilinx Vitis IDE – výběr platformy pro vytvoření Application Project.

V následujícím kroku je doporučeno vytvořit aplikaci z předpřipravených ukázkových souborů, ze kterých může být aplikace dále rozvinuta dle potřeb. Nejlepším řešením se autorovi jevilo využívat *Simple Vector Addition*. Při využití ukázkových souborů jsou již potřebné části projektu předkonfigurovány a je možné je vhodně upravovat dle požadavků aplikace.

Po načtení ukázkového projektu je možné spustit build proces přímo z Vitis IDE, nebo pomocí příkazové řádky. Při realizaci této práce bylo využíváno příkazové řádky, protože disponovala rychlejší odezvou na požadovaný build a snadněji se tak aplikace iterovala a vyvíjela.

Aby bylo možné build proces vyvolávat z příkazové řádky, je nutné vytvořit **makefile** soubory, které slouží ke komplikaci dílčích zdrojových souborů a k vyvolání **v++ linkeru**, jež spojí vytvořené dílčí soubory pro PL do jednoho binárního souboru **xclbin**. **Makefile** soubory je možné vytvořit manuálně, nebo pomocí Vitis IDE v okně *Assistant* poklikem pravým tlačítkem myši na název potřebné části a volbou *Create Makefiles*.

Volbu **makefiles** je potřeba provést pro složky:

1. `<application-project-name>_hw_link [HW Link]`,
2. `<application-project-name>_kernels [HW Kernel]`,
3. `<application-project-name> [Host]`,
4. `<type-of-build>`

(mimo podsložku, umístěno v hlavní složce <application-project-name> /System).

V některých případech nejsou **makefile** soubory vytvořeny, nebo aktualizovány dle posledních změn v projektu. Proto je nutné v okně *Assistant* pravým tlačítkem kliknout na jednotlivé složky a zvolit *Refresh Project Models* a v okně *Explorer* zvolit aplikaci a v nabídce po využití pravého tlačítka myši zvolit *Refresh* a proces generování **makefile** souborů opakovat.

Struktura **makefile** souborů je obecná a firma Xilinx dodává na svých stránkách dokumentaci k jejich tvorbě. V některých případech, kdy nedocházelo k jejich automatické aktualizaci, byl autor nucen tyto soubory manuálně upravit. Zejména se úpravy týkaly přidání nových souborů pro generování a linkování objektů pro kernel.

Doporučený postup build procesu pro *Hardware* build je následující:

1. build host aplikace pomocí **makefile** v cestě

<vitis-workspace>/<application-name>/Hardware/
(nejrychlejší proces),

2. build kernelu pomocí **makefile** v cestě

<vitis-workspace>/<application-name>_kernels/Hardware
(středně pomalý proces),

3. spojování objektů **xo** z kroku č. 2 do objektu **xclbin** pro PL pomocí **makefile** souboru v cestě
<vitis-workspace>/<application-name>_system_hw_link/Hardware/ (nejpo-
malejší proces, obsahuje syntézu, implementaci, generování bitstreamu), tento proces u vytváření
aplikace v některých případech trval i 1–2 hodiny.

Dle dostupných informací Xilinx Kria K26 SOM momentálně nepodporuje emulaci. Proto byl představen postup pro *Hardware* build, který je možné aktivovat otevřením konfiguračního souboru

<application-name>_kernels.prj v okně *Explorer*, který se nachází v cestě

<application-name>_system/<application-name>_kernels/, a výběrem **Hardware** v nastavení *Active build configuration*.

Je nutné upozornit, že tento soubor je opět upravitelný v textovém editoru a této skutečnosti bylo při realizaci aplikace velmi využíváno. Soubor obsahuje nastavení jednotlivých akcelerovaných funkcí, výběr optimalizace build procesu a nastavení šířky portu pro přenos dat mezi PS a PL akcelerovanou funkcí. V případě, že jsou přidávány do projektu další akcelerované funkce, pro jejich zkompilování je nutné zvolit odpovídající funkce v tomto souboru pomocí GUI nabídky. Ovšem v některých případech nedochází ke správné indexaci souborů ze strany Vitis IDE a je nutné upravit soubor v textovém editoru.

15 Deployment aplikace na platformu

Po úspěšném dokončení build a linking procesu aplikace ve Vitis, je možné přistoupit k procesu *deployment* (nasazení) aplikace do systému vývojové desky.

Z kroku *Konfigurace Device Tree* byl získán soubor `pl.dtbo`, který byl překopírován do složky `transfer`, definované v *Struktura složek*. Do téže složky je nyní možné překopírovat soubory aplikace:

- `<application-name>/Hardware/<application-name>` (host program),
- `<application-name>_system_hw_link/`
`Hardware/<binary-container>.xclbin` (PL bitstream),

které byly získány z build procesu pomocí Vitis.

Aby bylo možné využívat příkazy `xmutil`, které využívají DFX-MGR (daemon) [56] pro konfiguraci PL, loading a unloading bitstreamů, spravování data modelu daných bitstreamů apod. je třeba pro akcelerovanou aplikaci vytvořit metadata soubor `shell.json`. Podpora dalších formátů `shell_type`, než jen `XRT_FLAT` je ve vývoji. [51]

Soubor `shell.json` je opět vhodné vytvořit na vývojářském PC a přemístit do složky `transfer`.

```
1 {
2   "shell_type": "XRT_FLAT",
3   "num_slots": "1"
4 }
```

Kód 15 - 1 Metadata shell.json soubor pro xmutil.

Před přesunem souborů aplikace do vývojové desky obsahuje složka `transfer` soubory:

- `pl.dtbo` (Device Tree Blob Object),
- `shell.json` (metadata soubor),
- `<application-name>` (aplikace pro PS),
- `<binary-container.xclbin>` (bitstream pro PL).

Bylo zjištěno, že nejrychlejším a nejfektivnějším způsobem pro přenos souborů mezi systémem, na kterém je aplikace vyvíjena (či je třeba data z aplikace zpracovávat), a systémem vývojové desky (*PetaLinux*) je použití `scp` (secure copy) přes interface `eth0`.

Příkaz přenosu potřebných souborů pro běh akcelerované aplikace ze složky `transfer` do složky `/home/petalinux` na K26 je v kódu 15 - 2.

Dle dostupné dokumentace operačních systémů Linux a macOS bylo zjištěno, že pokud systém vývojového PC využívá OpenSSH verze 9.0 a vyšší, je pro přenos souborů pomocí `scp` výchozím protokolem SFTP. Bez dodatečného nastavení *PetaLinux* by tudíž nebylo možné pouze pomocí `scp` příkazu přenášet soubory. Proto je nutné využít tzv. *legacy scp protocol*, aktivovaný pomocí `-O` v příkazu `scp` (15 - 2).

```
1 scp -O pl.dtbo shell.json <application-name> <binary-container.xclbin>
      root@<ip-address-of-eth-interface>:/home/petalinux
```

Kód 15 - 2 Příkaz pro přesun souborů pomocí `scp` ze systému, kde byla aplikace vyvíjena do systému *PetaLinux* na K26 SOM.

Pro práci na vývojové desce v systému *PetaLinux* autor doporučuje využívat `ssh` a komunikaci přes USB/UART používat pouze jako prostředí pro výpis informací z *kernel ring bufferu*, které nejsou pomocí `ssh` bez použití `dmesg` uživateli viditelné.

Aby došlo k funkčnímu nakonfigurování PL pomocí bitstreamu, je třeba přejmenovat soubor `<binary-container.xclbin>` na `<binary-container.bin>` (např. pomocí příkazu `mv`). Pokud by nedošlo k přejmenování, byla by vypsána chybová hláška.

Pro spuštění akcelerované aplikace `dfx-mgr` vyžaduje, aby bitstream a soubory aplikace byly vloženy do cesty `/lib/firmware/<company-name>/<application-name>`. Do verze *PetaLinux* 2022.1 je podporován pouze název `xilinx` jako `<company-name>`, od verze 2022.1 jsou podporovány volitelné názvy, ovšem musí být vloženy do souboru `daemon.config`, umístěného v `/etc/dfx-mgrd/daemon.conf`. [51] [56]

Příkaz pro tvorbu složky `<application-name>` a pro kopírování požadovaných souborů je zobrazen v 15 - 3.

```
1 mkdir -p /lib/firmware/xilinx/<application-name>
2
3 cp pl.dtbo <application-name> <binary-container.bin> shell.json /lib/
firmware/xilinx/<application-name>
```

Kód 15 - 3 Příkaz vytvoření potřebné složky aplikace a kopírování potřebných souborů firmwaru.

Následně je možné provést unloading stávající aplikace a loading aplikace `<application-name>`. Potřebné příkazy s komentáři jsou zobrazeny v 15 - 4.

```
1 xmutil unloadapp # unloading currently active application
2
3 xmutil listapps # list currently available applications
4
5 xmutil loadapp <application-name> # load user defined application
```

Kód 15 - 4 Příkaz vytvoření složky aplikace a kopírování potřebných souborů firmwaru.

Po načtení aplikace je možné ve výpisu `dmesg` nebo USB/UART pozorovat, zda došlo ke správnému načtení bitstreamu, zařízení, přiřazení přerušení apod.

Aplikaci a bitstream které jsou aktuálně načtené je možné spustit i mimo `firmware` cestu. Autor doporučuje mít ponechané kopie aplikací v uživatelské složce `/home/petalinux/<application-name>` a spouštět aplikace příkazem uvedeným v 15 - 5.

```
1 ./<application-name> <binary-container.bin>
```

Kód 15 - 5 Příkaz pro spuštění akcelerované aplikace.

V této části byl představen doporučený postup deploymentu aplikace na zařízení Kria K26 SOM. Postup byl doplněn o užitečné příkazy a poznámky, které spuštění aplikace urychlují, usnadňují a nebo přinášejí možnost automatizace pomocí `bash` skriptů.

S určitou pravděpodobností existují i jiné způsoby deploymentu aplikací, ale autor je v době realizace této práce zatím nenalezl.

16 Popis pracoviště

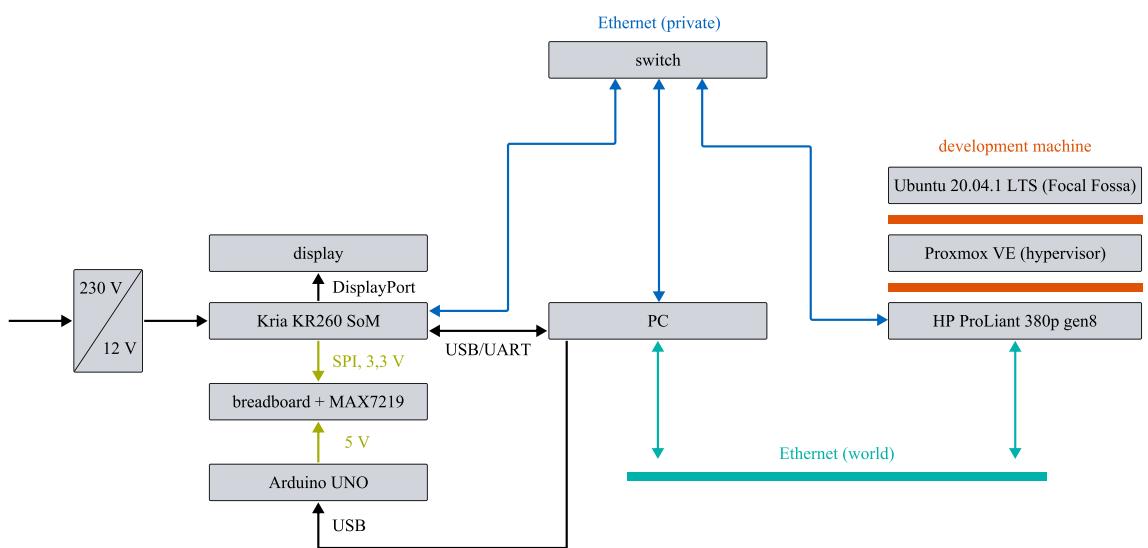
Na obr. 16 - 1 je zobrazeno blokové schéma pracoviště, které bylo vytvořeno v rámci této práce a slouží k vývoji akcelerovaných aplikací.

Blok *PC* představuje osobní počítač, který slouží k přístupu k pracovišti a je propojeno s vývojovou deskou KR260 pomocí *USB/UART* spojení, které je vhodné v případě, že nedošlo k inicializaci ethernet adaptéra. Vývojová deska je pomocí adaptéra *eth0* připojena do switche na pracovišti, ke kterému je uživatelský počítač také připojen. Pomocí **modrého značení** jsou do switche připojeny také bloky **development machine**. Fyzicky se jedná o připojení serveru pomocí adaptéra *eno2*, který je umístěn ve fakultním datacentru. Tímto způsobem je vytvořena virtuální síť, která je využívána pouze pro komunikaci mezi zařízeními vývojového pracoviště.

Připojení serveru pomocí adaptéra *enol* do externí sítě **Ethernet (world)** přináší krom možnosti vzdáleného přístupu do virtualizovaného zařízení v hypervisoru Proxmox VE také možnost vzdáleného přístupu do sítě **Ethernet (private)**. Využití serveru pro tvorbu aplikací bylo zvoleno z důvodu výkonové a časové náročnosti tvoření *PetaLinux* systému, HW designu ve Vivado a tvoření bitstreamu ve Vitis IDE. Díky vzdáleným přístupům je možné pracovat na vývoji i mimo laboratoř. Protože v některých případech při restartování vývojové desky nedochází ke správné inicializaci adaptéra *eth0*, je v navazující práci plánováno připojit vývojovou desku k SBC (single board computer) pomocí *USB/UART* rozhraní a SBC připojit do **Ethernet (private)**. K vývojové desce bude možné přistupovat cestou **Ethernet (world)** -> **development machine** -> **Ethernet (private)** -> PCB -> *USB/UART* -> Kria KR260. Toto fyzické spojení umožní přístup ke K26 pomocí *minicom*. Po připojení bude možné provést manuální inicializaci adaptéra po restartování vývojové desky.

Protože ukázka využití SPI komunikace je realizována pomocí obvodu *MAX7219* s LED maticí, který vyžaduje napájení v rozmezí 4 V až 5 V a výstupy PMOD dodávají napětí pouze 3,3 V, je třeba pro napájení testovacího rozhraní využít externího zdroje. V této práci byla pro napájení obvodu využita deska Arduino UNO, která je napájena z USB rozhraní uživatelského počítače. Volba tohoto napájení byla vybrána z důvodu jednoduchosti a dostupnosti. Provedení zapojení pro testování SPI komunikace je znázorněno v části *Zapojení pro testování SPI komunikace*.

Představená architektura pracoviště byla tvořena tak, aby nebylo problematické ji rozšiřovat o další prvky a umožňovala vzdálený přístup s podporou více aktivních uživatelů v jeden okamžik.



Obr. 16 - 1 Blokové schéma uspořádání vývojového pracoviště.

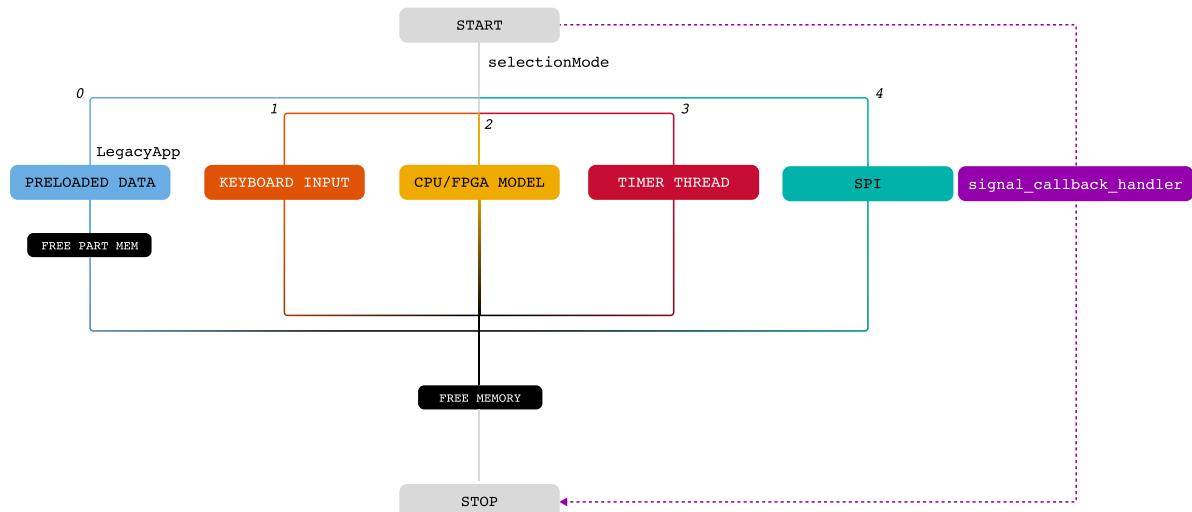
17 Vytvořená aplikace

V této části je představena vytvořená aplikace pro PS a akcelerovaná aplikace pro PL. Shodnocení výsledků aplikací je uvedeno v sekci *Poznátky získané profilováním aplikací*.

Hlavním cílem této práce bylo představit možnosti využití platformy pro řízení elektrického pohonu nebo systém HIL. Proto jednotlivé ukázkové podaplikace reprezentují části a algoritmy, které by bylo možné využít při realizaci skutečného připojení platformy k řízenému pracovišti.

Aby bylo možné představit jednotlivé funkce, které by byly používány při řízení elektrického pohonu, je ukázková aplikace rozdělena do pěti větví, které je možné vybrat po spuštění aplikace. Je nutné podotknout, že větve PRELOADED DATA bylo nutné realizovat do zvláštní aplikace, protože současné vkládání všech vytvořených kernelů do bitstreamu by vyžadovalo příliš mnoho resources (např. FIFO). Aby bylo možné všechny větve aplikace realizovat s pomocí pouze jednoho bitstreamu, bylo by nutné podrobit algoritmus kernelu další optimalizaci.

Základních pět větví ukázkové aplikace a jejich vztahy k celkovému běhu programu jsou zobrazeny na obr. 17 - 1.



Obr. 17 - 1 Základní větvení ukázkové aplikace.

Po spuštění hlavního procesu je pomocí `selectionMode` proměnné možné volit, jaká podaplikace bude spuštěna. Následně je provedena dynamická alokace paměti pro využívané struktury a pole v daných aplikacích. Je nutné zmínit, že PRELOADED DATA aplikace využívá unikátní sadu proměnných, které nejsou využívány v ostatních aplikacích. Z tohoto důvodu obsahuje PRELOADED DATA samostatnou skladbu instrukcí pro dealokaci paměti. Jednotlivé podaplikace jsou představeny v následujících částech práce pomocí názorných vývojových diagramů.

Po ukončení podaplikací jsou vyvolány příkazy pro dealokaci paměti, která byla alokována dynamicky v programu PS.

17.1 Bezpečnost při uživatelském ukončení aplikace

Z důvodu bezpečnosti je do algoritmu aplikace zakomponována funkce `signal_callback_handler`, jež reaguje na signál `SIGINT`. Tento signál je momentálně vysílán v případě přerušení běhu aplikace uživatelem. Handler funkce je registrována na signál již na počátku hlavní `main(int argc, char *argv[])` funkce aplikace.

V kódu č. 17 - 1 je představana deklarace struktury typu `InvertorSwitchType`, která slouží k uložení hodnot stavu virtuálních spínačů invertoru. Poté následuje deklarace funkce `stopInvertor()`, která po vyvolání nastaví stavy všech virtuálních spínačů na 0 a zajistí vypsání informační hlášky ohledně tohoto děje pro případ debugingu.

Funkce `signal_callback_handler` vypisuje hlášku o zachycení `SIGINT` signálu, zajišťuje spuštění funkce `stopInvertor()` a ukončení programu.

```
1 // main.cpp
2 InvertorSwitchType invertorSwitchGlobal;
3
4 void stopInvertor()
5 {
6     invertorSwitchGlobal.sw1 = 0;
7     invertorSwitchGlobal.sw2 = 0;
8     invertorSwitchGlobal.sw3 = 0;
9     invertorSwitchGlobal.sw4 = 0;
10    invertorSwitchGlobal.sw5 = 0;
11    invertorSwitchGlobal.sw6 = 0;
12
13    std::cout << "Set all invertor switches at 0!\n";
14 }
15
16
17 // Define the function to be called when ctrl-c (SIGINT) is sent to process
18 void signal_callback_handler(int signum)
19 {
20     std::cout << "\nCaught signal " << signum << "\n";
21
22     // stoping invertor of global variables
23     stopInvertor();
24     std::cout << "Terminating program!\n";
25     // Terminate program
26     exit(signum);
27 } // Define the function to be called when ctrl-c (SIGINT) is sent to
28 // process
29 void signal_callback_handler(int signum)
30 {
31     std::cout << "\nCaught signal " << signum << "\n";
32
33     // stoping invertor of global variables
34     stopInvertor();
35     std::cout << "Terminating program!\n";
36     // Terminate program
37     exit(signum);
38 }
39 // main.cpp
40 main(int argc, char *argv[])
```

```

41 {
42     // Register signal and signal handler
43     signal(SIGINT, signal_callback_handler);
44 .
45 ..
46 ...
47 }

```

Kód 17 - I signal_callback_handler funkce a její registrace na signál SIGINT.

17.2 Keyboard Input

Program, který je v podaplikaci *Keyboard Input* akcelerován v PL, se nazývá kernel. (v tomto kontextu není myšleno jádro systému Linux) Tento kernel je realizován v PL a jeho úkolem je provést časově náročné výpočty. V podaplikaci *Keyboard Input*, *CPU/FPGA Model* a *Preloaded Data* byl v PL realizován matematický *I-n* model asynchronního motoru, regulace a Space Vector Modulation (SVM). Hlavním výstupem kernelu jsou virtuální stavy sepnutí spínačů, které by v reálné aplikaci byly napojeny na fyzické piny PMOD a ovládaly by drivery řízených výkonových polovodičových prvků měniče.

Naznačený algoritmus aplikace je zobrazen na obr. 17 - 2. Akcelerované aplikace v této práci obsahují totožné inicializační a deinicializační funkce.

17.2.1 Konfigurace PL

Blok **PL SETTINGS PREAMBLE** obsahuje kód, jehož předloha byla získána z ukázkových souborů akcelerovaných aplikací přímo v aplikaci Vitis. V tomto bloku dochází k inicializaci PL zařízení, definování příkazové fronty (Command Queue), programu pro PL, platformy a k definici kernelů. Funkce a definice typů, jež jsou použity pro jednotlivé proměnné jsou pro akcelerované aplikace definovány v **c12.hpp** knihovně. V preambuli se nachází smyčka, jež vyhledává platformy zařízení Xilinx a pokud dojde k jejímu úspěšnému nalezení, pokouší se algoritmus do ní nahrát bitstream ve formátu *xclbin* nebo *bin*. Po úspěšném nahrání bitstreamu, jsou deklarovány potřebné elementy pro spouštění kernelu. (Context, CommandQueue, Program, Kernel). Veškeré funkce, jež využívají *openCL* knihovny, jsou uzavřeny do definovaného makra, které v případě, že vykonávaný příkaz navrátí jinou návratovou hodnotu než **CL_SUCCESS** (0), vypíše informace o dané chybě. Toto makro není podstatné pro funkční aplikaci, ovšem je velmi vhodné pro možný debugging. Definice makra byla také převzata z ukázkových akcelerovaných aplikací.

17.2.2 PS Aplikace

Pokud průchod aplikace částmi pro inicializaci a konfiguraci PL byl úspěšný, aplikace přechází do části, ve které jsou definovány a deklarovány parametry pro simulaci, pro *I-n* model motoru a pro regulaci. V části **PARAMETERS SETTINGS** dochází k dynamické alokaci paměti pomocí **posix_memalign()** funkce. U některých parameterů je při jejich definici znán jejich počet, proto by bylo možné použít statické alokace paměti. Ovšem pro demonstraci práce s **posix_memalign()** byl zvolen přístup dynamické alokace. Při použití dynamické alokace je nutné provést zarovnání paměti na 4096 bitů,

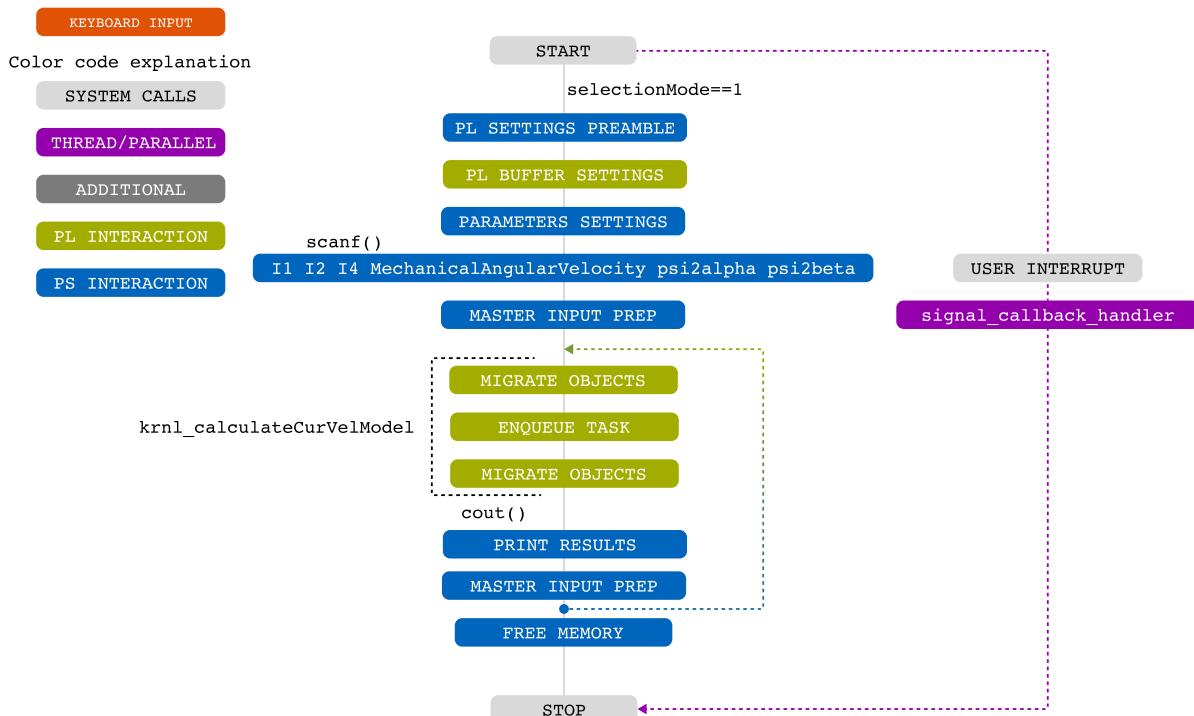
Následně je uživatel tázán o jakou podaplikaci má zájem, po zadání čísla, odpovídající dané podaplikaci **KEYBOARD INPUT** je uživatel tázán ohledně vstupních hodnot do *I-n*, které jsou pomocí funkce **scanf()** vloženy do odpovídajících proměnných.

V algoritmu je dále uveden blok **MASTER INPUT PREP**, který reprezentuje přípravu datového pole

typu *float*, které je využito k přesunu dat do globální paměti. Z globální paměti jsou data využívána akcelerovaným kernelem. Je možné přenéset do globální paměti i více vstupních proměnných, které budou ukládány do globální paměti, ovšem poté je v kernelu nutné vytvořit více interfaces. Větší množství rozhraní je třeba, aby bylo možné data načítat paralelně a urychlit tím dobu vykonávání kernelu. Při zkoumání využitelnosti platformy v této práci byl ovšem vliv rozdělení jedotlivých interfaces na rychlost vykonávání minimální. Vznikl však rozdíl ve využitých resources. Pokud se v platformě Digilent Zybo využilo více interfaces a bylo požadováno paralelní načítání dat kernelem, měl tento požadavek v některých případech za následek nemožnost umístění designu na PL, protože bylo překročeno maximálního počtu využitých resources.

17.2.3 Interakce s kernelem

Po přípravě proměnné `masterInput` je do CommandQueue pro dané zařízení vložen požadavek na přesun proměnných v podobě vytvořeného bufferu do globální paměti, přístupné PL. Po ukončení přenosu bufferů do paměti, je možné do CommandQueue zařadit požadavek na spuštění kernelu `krnl_calculateCurVelModel`. Pokud dojde k úspěšnému vykonání kernelu, je možné přesunout výsledné hodnoty uložené v bufferu z globální paměti do paměti PS pomocí příkazu `enqueueMigrateMemObjects`. Část reprezentující komunikaci s kernelem, ohraničená na obr. 17 - 2 přerušovanou křivkou, je zobrazena v kódu 17 - 2. Postup interakce s tímto kernelem je totožný i v podaplikacích *CPU/FPGA Model* a *Preloaded Data*. Interakce s kernelem `krnl_calculateInvMot` v podaplikaci *CPU/FPGA Model*, která využívá akcelerovaný model asynchronního motoru, je odlišná ve struktuře přenášeného bufferu do a z globální paměti.



Obr. 17 - 2 Keyboard Input větev aplikace – manuální nastavování vstupních hodnot do I-n modelu.

```

1 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_masterInput}, 0 /* 0 means from host*/));
2 OCL_CHECK(err, q.finish());

```

```

3
4 OCL_CHECK(err, err = q.enqueueTask(krnl_calculateCurVelModel));
5 OCL_CHECK(err, q.finish());
6
7 OCL_CHECK(err, q.enqueueMigrateMemObjects({buffer_masterOutput},
     CL_MIGRATE_MEM_OBJECT_HOST));
8 OCL_CHECK(err, q.finish());

```

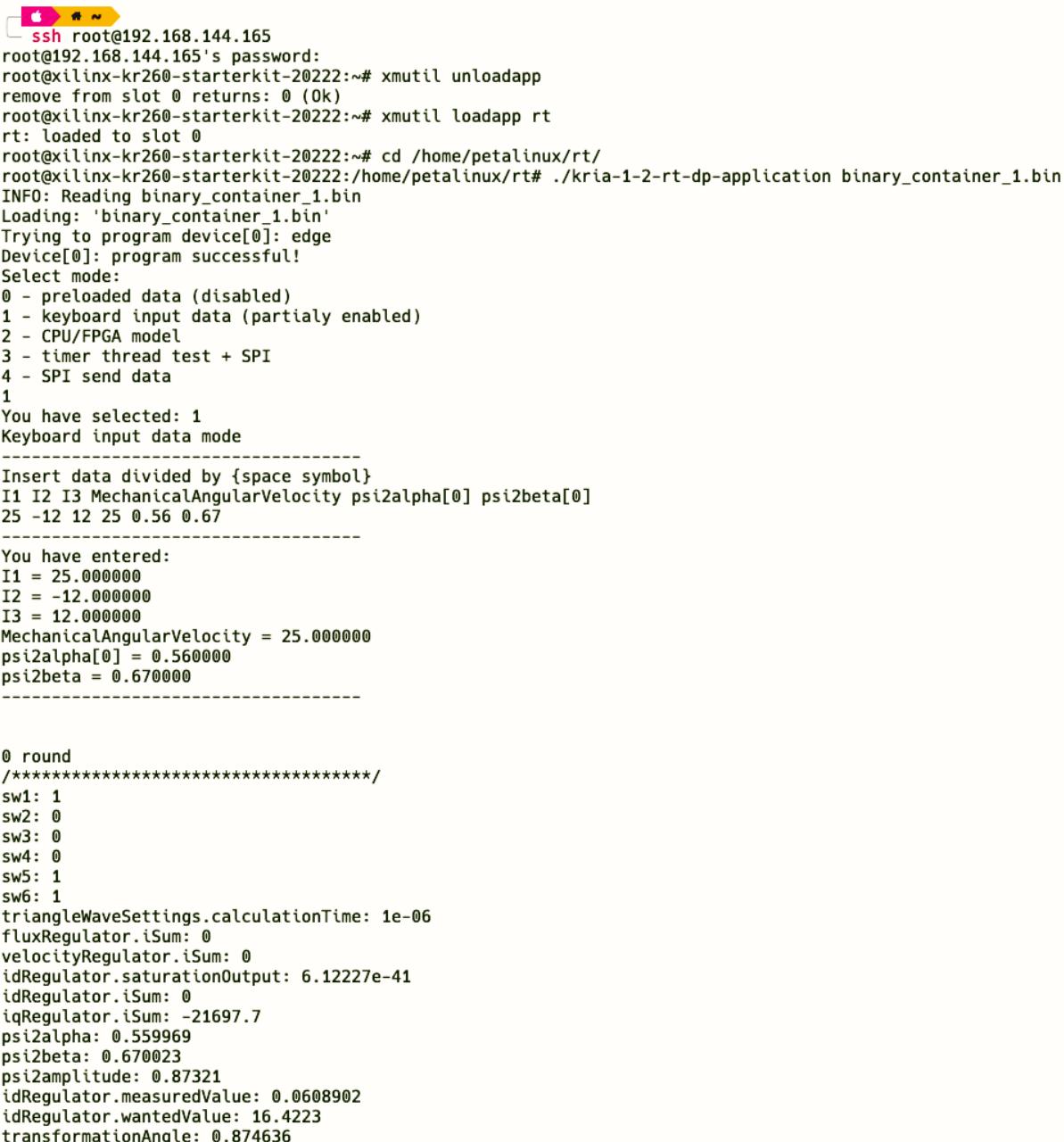
Kód 17 - 2 Interakce s kernelem krnl_calculateCurVelModel.

Po přesunu dat z globální paměti je vybraná sada hodnot proměnných vypsána uživateli v konzoli. Nyní by bylo možné program ukončit, ale pro demonstraci funkce aplikovaného RT, je znovu deklarován vstupní vektor proměnných a opět je kernel spuštěn. Pokud jsou výsledky získané z akcelerovaného kernelu totožné s předchozí iterací, došlo ke správné aplikaci RT patche.

Na počátku vývoje a debuggování aplikace, kdy nebyl RT patch využit, vznikal problém nekonzistentnosti výsledků mezi iteracemi. Více o problematice RT patche je uvedeno v části *RealTime Linux Patch*.

Na závěr dojde k odmapování bufferů od přidružených ukazatelů, které byly využity pro interakci s kernelem. Program je zakončen uvolněním využité paměti, dynamicky alokované v předchozích krocích.

V obr. 17 - 3 je možné pozorovat snímek konzole/terminálu, který zachycuje postup přihlášení pomocí ssh k vývojové desce s K26, unload stávající aplikace, loading uživatelské aplikace rt a její následné spuštění. Dále je zobrazen výběr podaplikace a výpis vybraných hodnot proměnných po první iteraci kernelu.



```

ssh root@192.168.144.165
root@192.168.144.165's password:
root@xilinx-kr260-starterkit-20222:~# xmutil unloadapp
remove from slot 0 returns: 0 (Ok)
root@xilinx-kr260-starterkit-20222:~# xmutil loadapp rt
rt: loaded to slot 0
root@xilinx-kr260-starterkit-20222:~# cd /home/petalinux/rt/
root@xilinx-kr260-starterkit-20222:/home/petalinux/rt# ./kria-1-2-rt-dp-application binary_container_1.bin
INFO: Reading binary_container_1.bin
Loading: 'binary_container_1.bin'
Trying to program device[0]: edge
Device[0]: program successful!
Select mode:
0 - preloaded data (disabled)
1 - keyboard input data (partialy enabled)
2 - CPU/FPGA model
3 - timer thread test + SPI
4 - SPI send data
1
You have selected: 1
Keyboard input data mode
-----
Insert data divided by {space symbol}
I1 I2 I3 MechanicalAngularVelocity psi2alpha[0] psi2beta[0]
25 -12 12 25 0.56 0.67
-----
You have entered:
I1 = 25.000000
I2 = -12.000000
I3 = 12.000000
MechanicalAngularVelocity = 25.000000
psi2alpha[0] = 0.560000
psi2beta = 0.670000
-----
0 round
*****
sw1: 1
sw2: 0
sw3: 0
sw4: 0
sw5: 1
sw6: 1
triangleWaveSettings.calculationTime: 1e-06
fluxRegulator.iSum: 0
velocityRegulator.iSum: 0
idRegulator.saturationOutput: 6.12227e-41
idRegulator.iSum: 0
iqRegulator.iSum: -21697.7
psi2alpha: 0.559969
psi2beta: 0.670023
psi2amplitude: 0.87321
idRegulator.measuredValue: 0.0608902
idRegulator.wantedValue: 16.4223
transformationAngle: 0.874636

```

Obr. 17 - 3 Snímek obrazovky konzole, zobrazující postup spuštění pod aplikace Keyboard Input a výpis první iterace výsledků z kernelu.

17.3 CPU/FPGA Model

Nejrozsáhlejší částí hlavního programu je podaplikace s názvem CPU/FPGA Model, která využívá dvou kernelů, které komunikují přes PS. Při prozkoumávání využití platformy autor narazil na možnost, že by kernely streamovaly data přímo mezi sebou, bez nutnosti použití PS. Tento způsob přenosu dat nebyl v této práci využit, ale je plánováno jeho testování. Tyto kernely by bylo také vhodné realizovat stylem *free running*, kdy by data byla z PS streamovaná přímo do jednotlivých kernelů v okamžiku, když by byla v PS k dispozici. To by nejspíše umožnilo rychlejší reakci kernelu, která by přinesla zkrácení doby potřebné na přesun dat z PS do PL a naopak. Experimentálně zjištěné časy a timeline graf je uveden v sekci 18.2

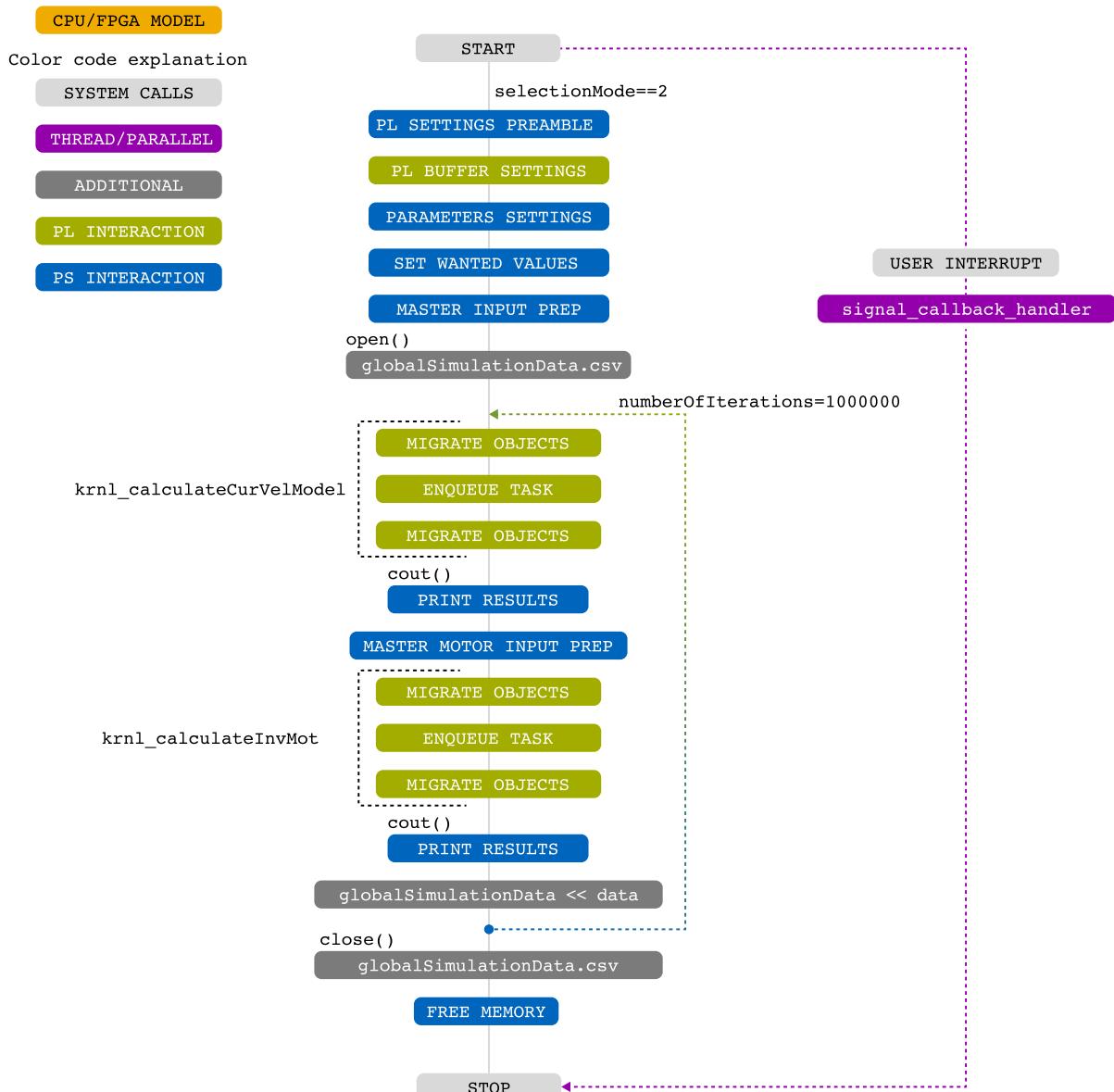
Algoritmus, využívaný v této práci, je uveden na obr. 17 - 4.

K inicializaci PL je využita totožná preambule jako v *Keyboard Input* podaplikaci. Stejným způsobem dochází i k inicializaci potřebných bufferů a k nastavení parametrů modelů a simulace.

Protože dochází k modelování i regulátorů, je v případě této podaplikace důležité nastavit žádané hodnoty regulovaných veličin. V této ukázkové aplikaci jsou žádané hodnoty mechanické otáčivé rychlosti Ω a velikosti magnetického toku rotoru ψ_2 nastaveny přímo v kódu aplikace. V případě realizace skutečné regulace by bylo vhodné disponovat možností měnit tyto hodnoty v průběhu běhu programu pomocí konzole, nebo pomocí externího zařízení, komunikující s platformou pomocí SPI nebo jiného druhu komunikace.

Před spuštěním smyčky, která zajišťuje hlavní iteraci programu, je v programu zařazeno vytvoření souboru `globalSimulationData.csv`, který slouží k uchování hodnot vybraných sledovaných veličin. Soubor je dále využíván při tvorbě výsledných grafů simulace pomocí Python skriptu. Výsledný graf simulace a důvod využití skriptu je uveden v části *Zobrazení výsledků simulace*.

V programu následuje blok označený `krl1_calculateCurVelModel`, jež rezprezentuje interakci s kernelem. Po migraci hodnot veličin z globální paměti do paměti PS je možné získané výsledky vypsat do konzole. Po provedení kernelu, obsahující *I-n* model, bloky regulace a algoritmus SVM, jsou získané stavy virtuálních spínačů vloženy do vstupního vektoru pro kernel `krl1_calculateInvMot`. Tento kernel obsahuje modely invertoru a asynchronního motoru, popsaného v části *Model stroje*. Po úspěšném provedení kernelu a získání vypočtených hodnot z globální paměti jsou hodnoty některých významných veličin pro čtyři první iteraci algoritmu vypsány v konzoli. Vybrané veličiny, u kterých je vhodné sledovat jejich závislost na čase, jsou vloženy do souboru `globalSimulationData.csv` pro další zpracování. Následuje uzavření aktivního souboru a dealokace paměti použitých proměnných a bufferů.



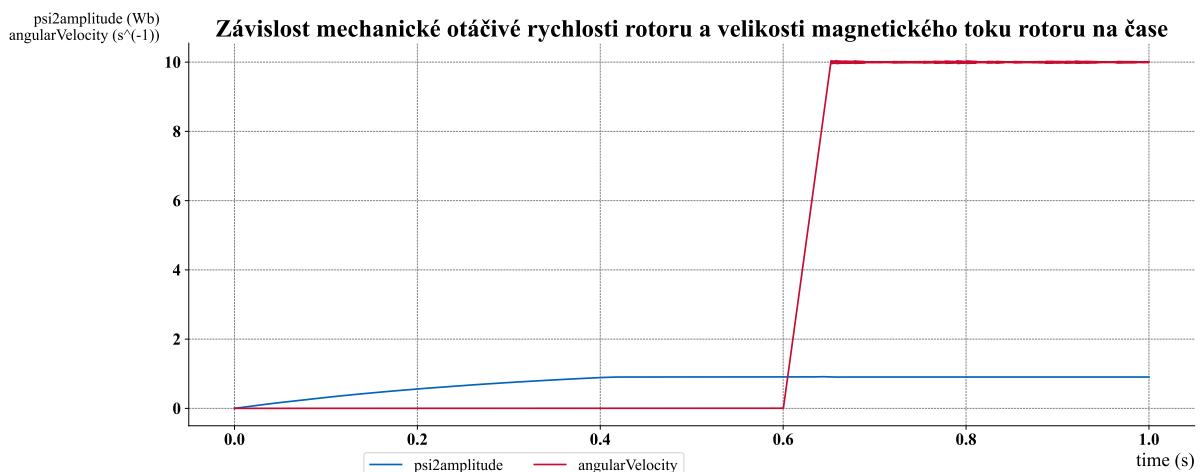
Obr. 17 - 4 CPU/FPGA Model větví aplikace – matematický I-n model, regulace, ASM model v FPGA.

17.3.1 Zobrazení výsledků simulace

Aby bylo možné při debuggingu aplikace vizuálně pozorovat výsledky simulace, bylo nutné data ze souboru `globalSimulationData.csv` vizualizovat způsobem, jež umožňuje rychlé zpracování velikého množství dat a v nejlepším případě je Open Source.

První přístup spočíval ke kontrole výsledků v programu Wolfram Mathematica. Ovšem při kroku simulace $1 \mu\text{s}$ a simulovaném čase 1 s se jednalo o výpis jednoho milion hodnot. Vykreslení takového množství bodů, které byly přímkově spojeny, bylo příliš náročné pro zařízení autora a trvalo příliš dlouho dobu. A protože Wolfram Mathematica vyžaduje licenci, bylo od tohoto způsobu upuštěno. Další možností bylo využití programu MATLAB™, ovšem opět je vyžadována licence a autor dbal na co největší otevřenost projektu. Proto bylo nakonec využito jazyku Python, jež není zatížen problémem *Vendor lock-in* a vykreslení požadovaného grafu se stejnou kvalitou jako v MATLAB™u nebo ve Wolfram Mathematica trvalo velmi krátkou dobu. Python skript pro vykreslení grafu je součástí přílohy této práce. Graf č. 17 - 5 reprezentuje výsledky simulace řízení asynchronního motoru pomocí FOC. Zobrazené výsledky odpovídají žádaným hodnotám, regulovaných veličin. V simulaci byly nastaveny požadované hodnoty:

- velikost magnetického toku rotoru $\psi_2 = 0,9032 \text{ Wb}$,
- velikost mechanické otáčivé rychlosti $\Omega = 10 \text{ s}^{-1}$ v čase $t = 0,6 \text{ s}$.



Obr. 17 - 5 Časová závislost velikosti mechanické otáčivé rychlosti Ω a velikosti magnetického toku rotoru ψ_2 . Výsledné hodnoty získané ze simulace akcelerované pomocí kernelu v PL.

17.4 Preloaded Data

Podaplikace preloaded data je z důvodů, popsaných v sekci *Vytvořená aplikace*, realizována v odděleném Application projektu v programu Vitis.

Aplikace byla součástí prvních experimentů s využitím kernelů a proto obsahuje starší (legacy) strukturu algoritmu, včetně legacy kódu v kernelu, který využívá více vstupních a výstupních vektorů a tudíž i interfaces (z důvodu testování optimalizace).

Zjednodušený vývojový diagram aplikace je na obr. 17 - 6. Stejně jako pro ostatní představené akcelerované aplikace je na počátku hlavního programu umístěn blok zajišťující inicializaci prvků pro kernel. Hlavním rozdílem od předchozích aplikací je definice a deklarace pouze jediného kernelu `krnl_CurVelLoadLegacy`. Ukázka deklarace je v kódu 17 - 3.

```
1 OCL_CHECK(err, krnl_CurVelLoadLegacy = cl::Kernel(  
2     program, "krnl_CurVelLoadLegacy", &err));
```

Kód 17 - 3 Deklarace kernelu krnl_CurVelLoadLegacy v podaplikace Preloaded Data.

Po potřebných deklaracích a definicích, nutných pro správnou funkci kernelu, je dynamicky naalokována paměť pro vybrané konstatní parametry simulace. Tyto parametry jsou např. krok simulace, časové rozmezí simulace či počet kroků. Protože je v podaplikaci simuloval zjednodušený $I-n$ model, kdy není uvažována změna parametrů stroje během simulace (např. rezistivita rotorového vinutí), jsou v bloku **PARAMETERS SETTINGS** definovány také parametry stroje vstupující do myšleného modelu.

Do této aplikace vstupují předpočítané nebo změřené hodnoty následujících veličin:

- proud I1, jež reprezentuje velikost proudu statoru $i_{1a}(t)$ procházející první fází do řízeného motoru,
- proud I2, jež reprezentuje velikost proudu statoru $i_{1b}(t)$ procházející druhou fází do řízeného motoru,
- proud I3, jež reprezentuje velikost proudu statoru $i_{1c}(t)$ procházející třetí fází do řízeného motoru,
- mechanická otáčivá rychlosť rotoru mototu Ω .

Uvedené hodnoty jsou načítány ze souboru **outputData.csv**, který je strukturován dle formátu uvedeného v 17 - 4. Formát nevyužívá nových řádků pro oddělení sady dat, pouze využívá oddělení pomocí čárky. Tento formát byl zvolen z důvodu, aby byla zajištěna co nejmenší velikost souboru a aby bylo maximálně zjednodušeno načítání dat pomocí funkce **std::getline**.

```
1 time,i1,i2,i3,MechanicalAngularVelocity,...,...
```

Kód 17 - 4 Struktura souboru outputData, ze kterého jsou načítány hodnoty pro akcelerovaný kernel.

Po načtení hodnot do definovaných proměnných jsou vytvořeny v PS buffery, které jsou mapovány na adresy odpovídajících ukazatelů a mají rozsah jednotlivých proměnných, ke kterým jsou mapovány. Buffery jsou nastaveny jako jednotlivé vstupní parametry kernelu.

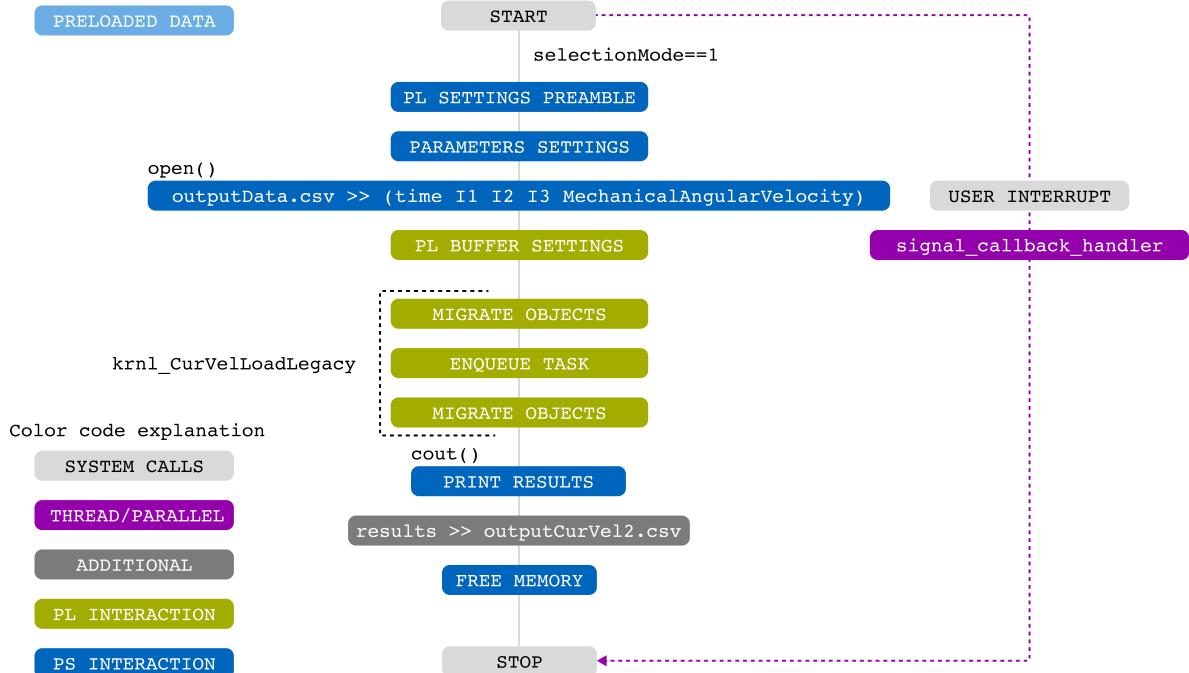
Ve vyznačené části krnl_CurVelLoadLegacy na obr. 17 - 6 je provedena sada příkazů pro přenos hodnot proměnných pomocí bufferů do globální paměti, přístupné z PL, spuštění kernelu a opětovné přenesení výsledných hodnot z globální paměti do PS.

V této legacy aplikaci je při výpisu hodnot vybraných veličin také provedeno ukládání hodnot do csv souboru **outputCurVel2.csv**. Struktura tohoto legacy souboru je v kódu 17 - 5.

Po ukončení podaplikace je opět dealokována paměť proměnných, která byla dynamicky alokována.

```
1 time,psi2Amplitude,transformAngle  
2 .  
3 ..  
4 ...
```

Kód 17 - 5 Struktura souboru outputCurVel2.csv, do něhož jsou umisťovány výstupní hodnoty vybraných veličin, vypočtených pomocí akcelerované aplikace.



Obr. 17 - 6 Preloaded Data větev aplikace – automatické čtení změřených/simulovaných vstupních hodnot do I-n modelu a jejich hromadné zpracování v kernelu.

17.5 SPI

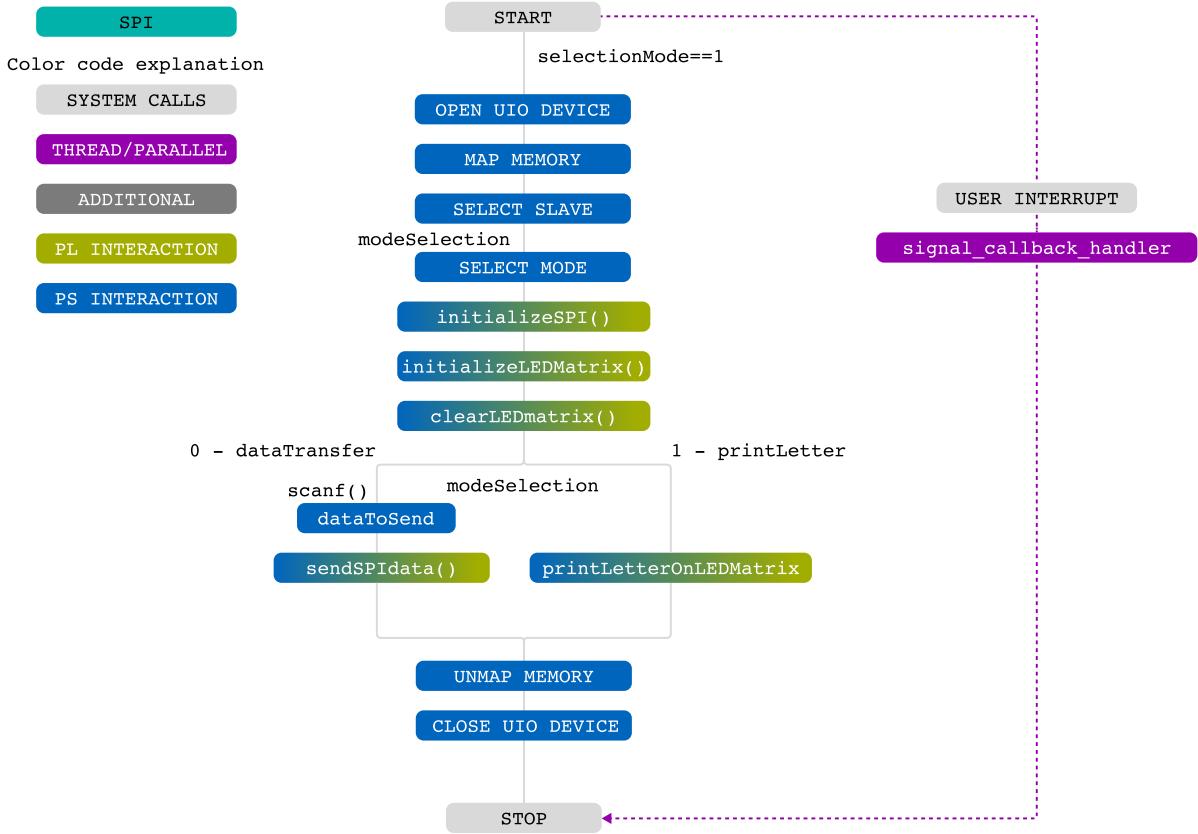
Pro potencionální komunikaci s externími jednotkami v reálné aplikaci bylo zvoleno SPI rozhraní. SPI bylo zvoleno, protože je využíváno externími jednotkami jako jsou ADC, DAC. Tyto jednotky jsou schopny díky SPI komunikovat s řídící platformou téměř v reálném čase.

Část aplikace, která prezentuje funkčnost SPI komunikace není oproti předchozím podaplikacím akcelerována. Tudíž nevyužívá akcelerované funkce (kernelu) pro urychlení výpočtů. Využívá však PL, ve kterém byl vytvořen IP blok *AXI Quad SPI*, který umožnuje SPI komunikaci. Tento blok způsobí vytvoření adres pamětí, které je možný z userspace programu nebo běhu *PetaLinux* systému adresovat.

Jiná možná realizace SPI komunikace by byla pomocí SPI driveru v PS, který by bylo nutné aktivovat při tvorbě *PetaLinux* systému.

Je důležité zmínit, že nakonfigurované parametry bloku *AXI Quad SPI* lze změnit pouze ve Vivado, tudíž je důležité při tvorbě HW platformy již znát požadavky zařízení, se kterými bude aplikace komunikovat. Tyto parametry jsou délka přenášené zprávy nebo frekvenci signálu CLK.

Pro zjednodušení adresování SPI registrů jsou v aplikaci definovány posuny od základní adresy pomocí `define` direktiv. Definice direktiv dle dokumentace [52] je zobrazena v 17 - 6.



Obr. 17 - 7 SPI větev aplikace – manuální odesílání dat přes SPI.

```

1 #define MAP_SIZE 0x10000
2 #define SPI_SPICR 0x60
3 #define SPI_SPISR 0x64
4 #define SPI_DTR 0x68
5 #define SPI_DRR 0x6C
6 #define SPI_SPISSR 0x70
7 #define SPI_TRANSMIT_FIFO_OCUPANCY 0x74
8 #define SPI_RECEIVE_FIFO_OCUPANCY 0x78
9 #define SPI_DGIER 0x1C
10 #define SPI_IPISR 0x20
11 #define SPI_IPIER 0x28

```

Kód 17 - 6 Definice posuvu adres pro registry AXI Quad SPI bloku pomocí define direktiv.

Jak je možné na názorném vývojovém diagramu 17 - 7 pozorovat, je pro zvýšení bezpečnosti programu využito mapování pouze potřebné části HW registrů. K mapování do uživatelského prostředí dochází pomocí UIO zařízení a `mmap` funkce. V DT jednotky je využit *generic-lio* driver, díky kterému je možné pozorovat vzniklá přerušení od SPI a tyto přerušení také znova povolovat pomocí acknowledge operace. Autor zatím nenašel způsob, pomocí něhož by bylo možné vykonat acknowledge přerušení v *PetaLinux*, pokud by bylo využito výchozích driverů SPI jednotky.

Po provedení mapování adres registrů do userspace pomocí funkce `initializeSPI()` (jejíž algoritmus je zobrazen v kódu 17 - 7) následuje inicializace jednotky, se kterou aplikace má komunikovat. Pro ukázkou využití SPI komunikace s externím prvkem byla realizována komunikace s jednotkou *MAX7219*,

připojenou na LED matici 8x8. Pokud by se jednalo o aplikaci v produkčním prostředí elektrických polohů, probíhala by komunikace nejspíše s analogově-digitálními převodníky, digitálně-analogovými převodníky, či jinými dostupnými jednotkami.

V ukázkové aplikaci je odeslána sekvence dat do *MAX7219* která zajistí, aby po její inicializaci došlo ke zhasnutí všech diod na matici. Tyto kroky jsou provedeny pomocí funkcí `initializeLEDMatrix()` a `clearLEDmatrix()`. Kód funkcí je součástí souborů, jež jsou přílohou této práce.

Po inicializaci je možné pomocí proměnné `dataTransfer` vybrat ze dvou testovacích módů. Mód při `dataTransfer == 0` je určen pro manuální odesílání zpráv pomocí SPI jednotky, mód kdy platí `dataTransfer == 1` odešle předem definované zprávy.

Ukázka testovacího zapojení vývojové platformy s jednotkou *MAX7219* je uvedena v části *Zapojení pro testování SPI komunikace*

Před ukončením aplikace jsou odmapovány HW paměti a je uzavřeno UIO zařízení.

```

1  /*
2   * @name      initializeSPI
3   * @brief     Initialize SPI communication in PL for desired slave and
4   *            mapped device.
5   * @todo      Implement interrupt and delete sleep_for...
6   */
7
8  void spiClass::initializeSPI(void *ptr, off_t slaveSelect)
9 {
10    *((unsigned *) (ptr + SPI_SPICR)) = 0x1E6;
11
12    *((unsigned *) (ptr + SPI_DTR)) = 0x06;
13
14    *((unsigned *) (ptr + SPI_SPISSR)) = 0x00;
15
16    *((unsigned *) (ptr + SPI_SPISSR)) = slaveSelect;
17
18    // Interrupt settings
19    // Global Interrupt Enable
20    *((unsigned *) (ptr + SPI_DGIER)) = 0x80000000;
21
22    // Interrupt enables
23    // *((unsigned *) (ptr + SPI_IPIER)) = 0x3FFF;           //
24    // enabling all interrupts
25    *((unsigned *) (ptr + SPI_IPIER)) = 0x4; // enabling only DTR is clear
26    // INT
27
28    off_t interruptStatus;
29
30    interruptStatus = *((unsigned *) (ptr + SPI_IPISR));
31    std::this_thread::sleep_for(std::chrono::nanoseconds(1)); // could
32    // not resolve other way now, because reading and writing to register takes
33    // more time than one tick probably

```

```

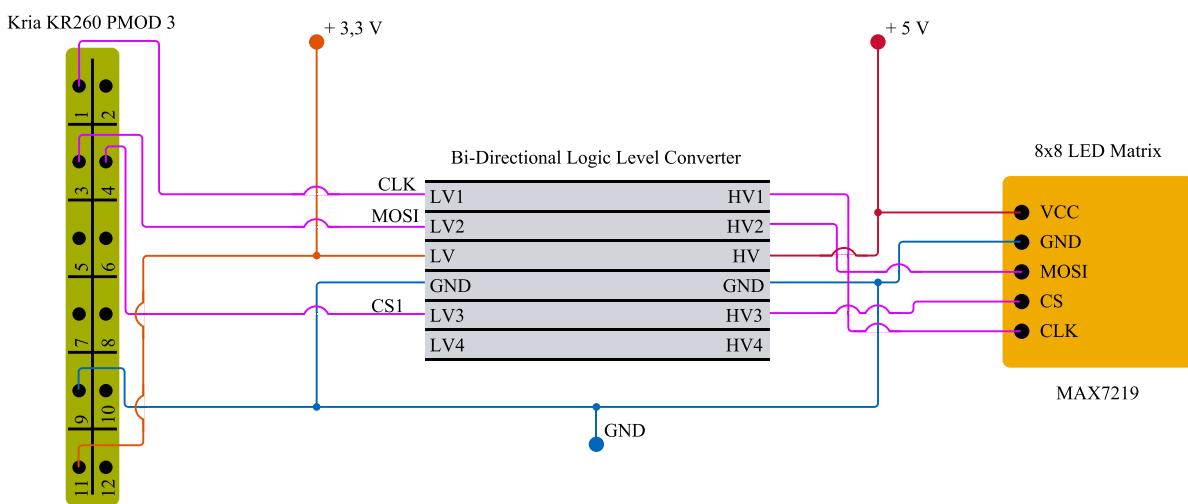
30     *((unsigned *) (ptr + SPI_IPISR)) = interruptStatus;      // 
resetting SPI interrupt status register
31 }

```

Kód 17 - 7 Algoritmus inicializace AXI Quad SPI jednotky v C++.

17.5.1 Zapojení pro testování SPI komunikace

Jak již bylo zmíněno v předchozích částech textu, v této práci je realizována ukázka využití SPI komunikace s pomocí obvodu *MAX7219*, který je připojen k LED matici. Obvod *MAX7219* vyžaduje napájecí napětí v rozmezí 4 V až 5 V. Výstupní napětí PMOD konektorů, dostupných na vývojové desce Kria KR260, je pouze 3,3 V. Aby bylo možné tento obvod využít k demonstraci funkce SPI, je pro připojení PMOD konektorů k obvodu *MAX7219* využit převodník úrovní. Zapojení pro testování SPI komunikace je zobrazeno na obr. 17 - 8.



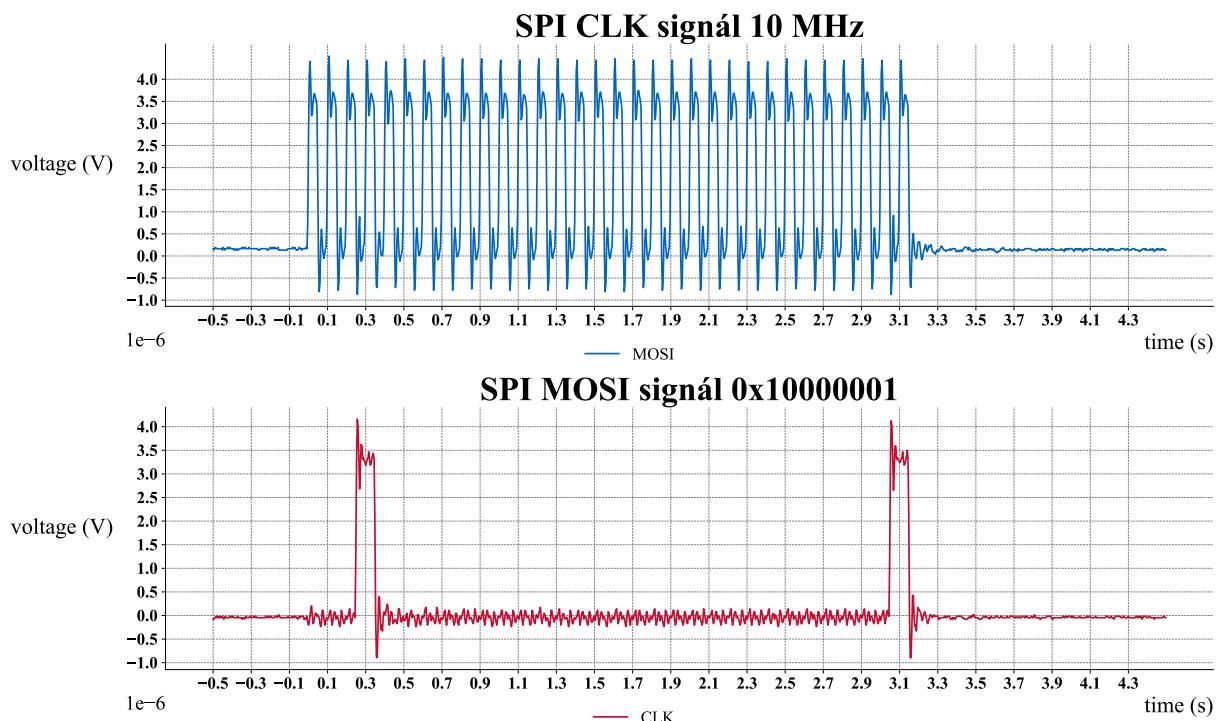
Obr. 17 - 8 Názorné schéma připojení PMOD3 Xilinx Kria KR260 k LED Matici s obvodem MAX7219.

V rámci prozkoumávání funkčnosti SPI komunikace byly pomocí osciloskopu zaznamenány osciloskopogramy pro signály CLK a MOSI.

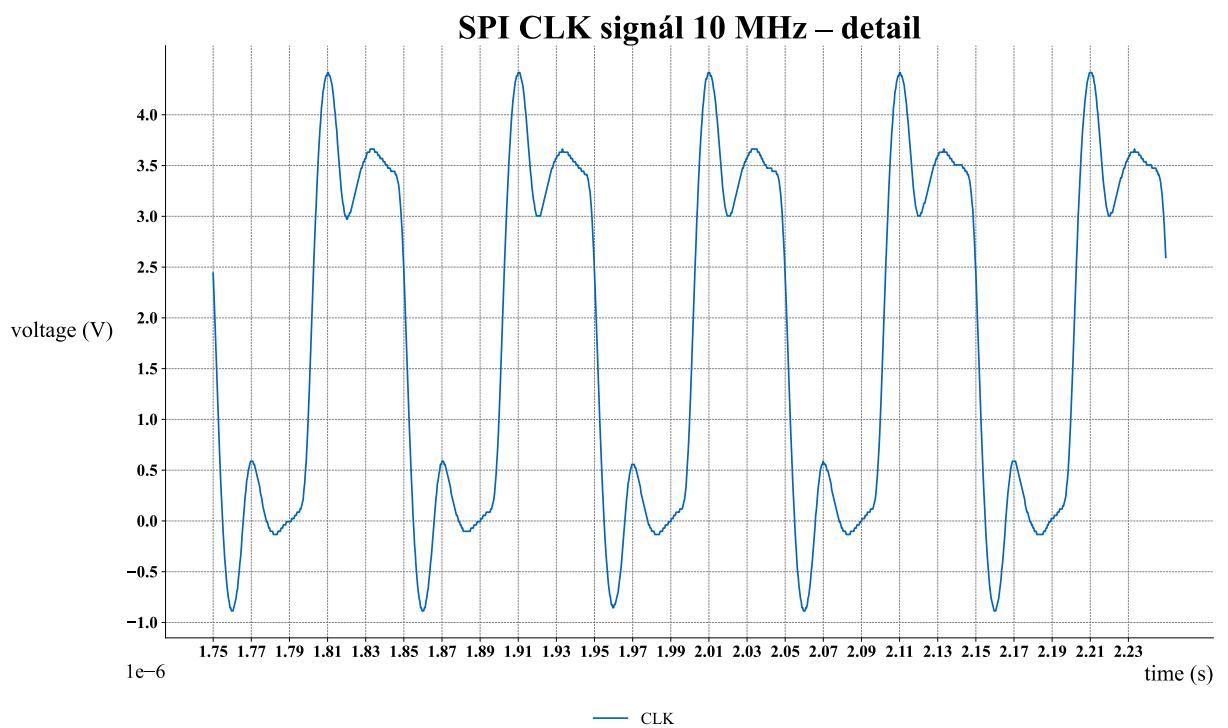
Na grafu 17 - 9 je možné pozorovat průběhy obou signálů a jejich vzájemnou závislost. Je vidět, že náběžná hrana signálu MOSI začíná před náběžnou hranou CLK. Tato souslednost je vytvořena z důvodu, aby již v okamžiku, kdy vzniká náběžná hrana CLK signálu, byla úroveň signálu MOSI ustálena na odpovídající hodnotě (3,3 V nebo 0 V).

Na obou signálech je vidět, že nedochází při náběžné hraně k dosáhnutí přesně úrovně 3,3 V, ale k překmitnutí hodnoty až na úroveň téměř 4,384 V. Při sestupné hraně dochází k podkmitnutí hodnoty 0 V na hodnotu -0,855 V. V případě komunikace s některými prvky nejsou tyto překmity žádoucí a bylo by třeba provést kroky k jejich odstranění. Důvod nedodržení úrovní by bylo vhodné více prozkoumat, ovšem tento úkol by byl nad rámec této práce a je možné mu věnovat úsilí v nadcházejících pracích.

V okamžiku neodesílání MOSI bylo pomocí osciloskopu 17 - 9 zjištěno, že se v signálu objevuje šum v napěťovém rozmezí přibližně -0,200 V až 0,170 V. Po ukončení přenosu signálu CLK je úroveň šumu již značně nižší. Je tudíž možné předpokládat, že rušení je způsobeno šířením elektromagnetického pole vodiče, přenášejícího CLK signál, prostorem kolem vodiče, který nezaručuje dostatečné odstínění možného EMC rušení.



Obr. 17 - 9 Osciloskop znázorňující časové průběhy SPI komunikace – 10 MHz CLK signál a MOSI signál přenášející data 0x10000001.



Obr. 17 - 10 Osciloskop znázorňující detail časového průběhu SPI komunikace – 10 MHz CLK signál.

17.6 Timer Thread

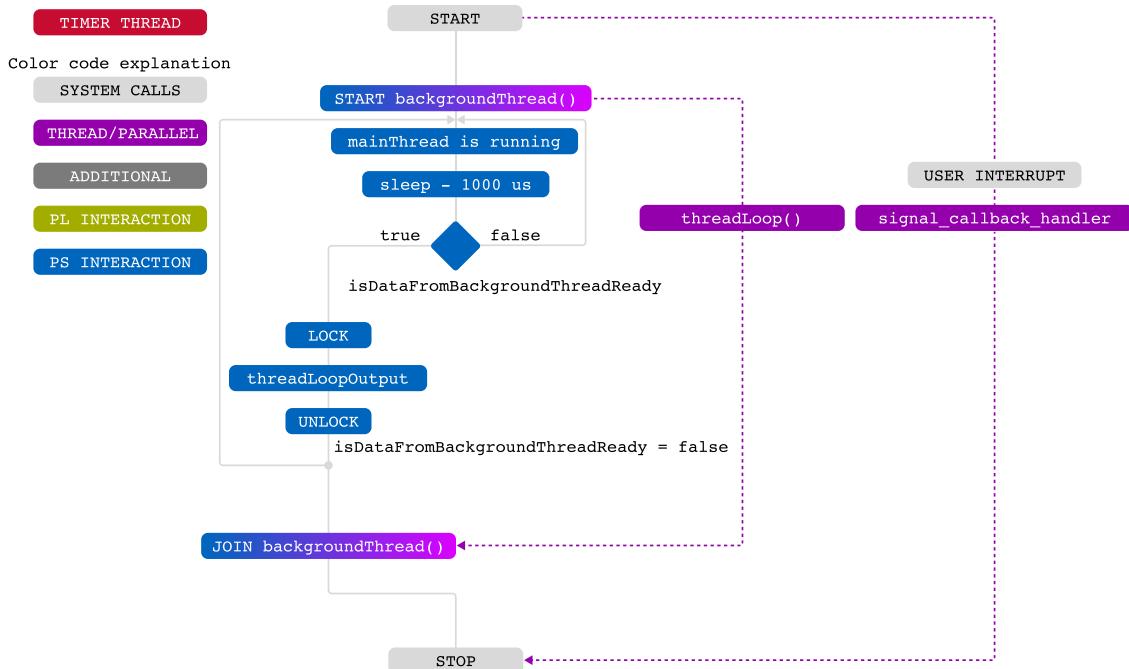
Timer Thread je opět podaplikace, která nevyužívá PL k akceleraci, ale k implementaci IP HW jednotek. V podaplikaci je využito IP bloku AXI Timer pro časování periodického děje. V ukázkové aplikaci je periodickým dějem spouštění funkce měnící obsah proměnné `threadLoopOutput` a spouštění funkce odesílání dat pomocí SPI jednotky. Pomocí SPI jednotky jsou odeslány do obvodu *MAX7219* data, která způsobují zobrazování dvou předem definovaných písmen na LED matici.

Tento ukázkový algoritmus má za úkol reprezentovat periodické získávání dat z analogově-digitálních převodníků a jejich následné zpracování v hlavní funkci. Jak je na vývojovém diagramu 17 - 11 možné pozorovat, pro obhospodařování AXI Timeru a SPI jednotky je vytvořeno samostatné vlákno, jež krom inicializační části obsahuje část výkonnou, která je uzavřena do nekonečné smyčky.

V reálné aplikaci řízení elektrického pohonu je plánováno využít tolik vláken, kolik bude potřeba paralelně získávat informací z ADC, které budou napojeny na proudové senzory a na senzor otáček.

Aby z vedlejších vláken do hlavního vlákna byla přenesena pouze validní data, je využito `mutex` třídy z knihovny `std` a logického semaforu, který umožní předání dat pouze v případě, že došlo ke kompletnímu uložení potřebné hodnoty do proměnné, ze které je hodnota čtena v hlavním vlákně. V případě realizace více paralelních vláken by bylo nutné zakomponovat podmínu, která by kontrolovala validitu všech paralelně čtených hodnot.

Algoritmy, provedené ve vedlejším vlákně, jsou představeny v části *Thread Loop*.



Obr. 17 - 11 SPI Timer Thread větví aplikace – AXI Timer Thread s automatickým odesíláním dat.

17.6.1 Thread Loop

Část aplikace Thread Loop, jejíž vývojový diagram je na obr. 17 - 12, reprezentuje soubor algoritmů a funkcí, které jsou prováděny ve vlákně `backgroundThread`.

Funkce, která je pomocí `backgroundThread` spuštěna, je v kódu nazvána `threadLoop()`.

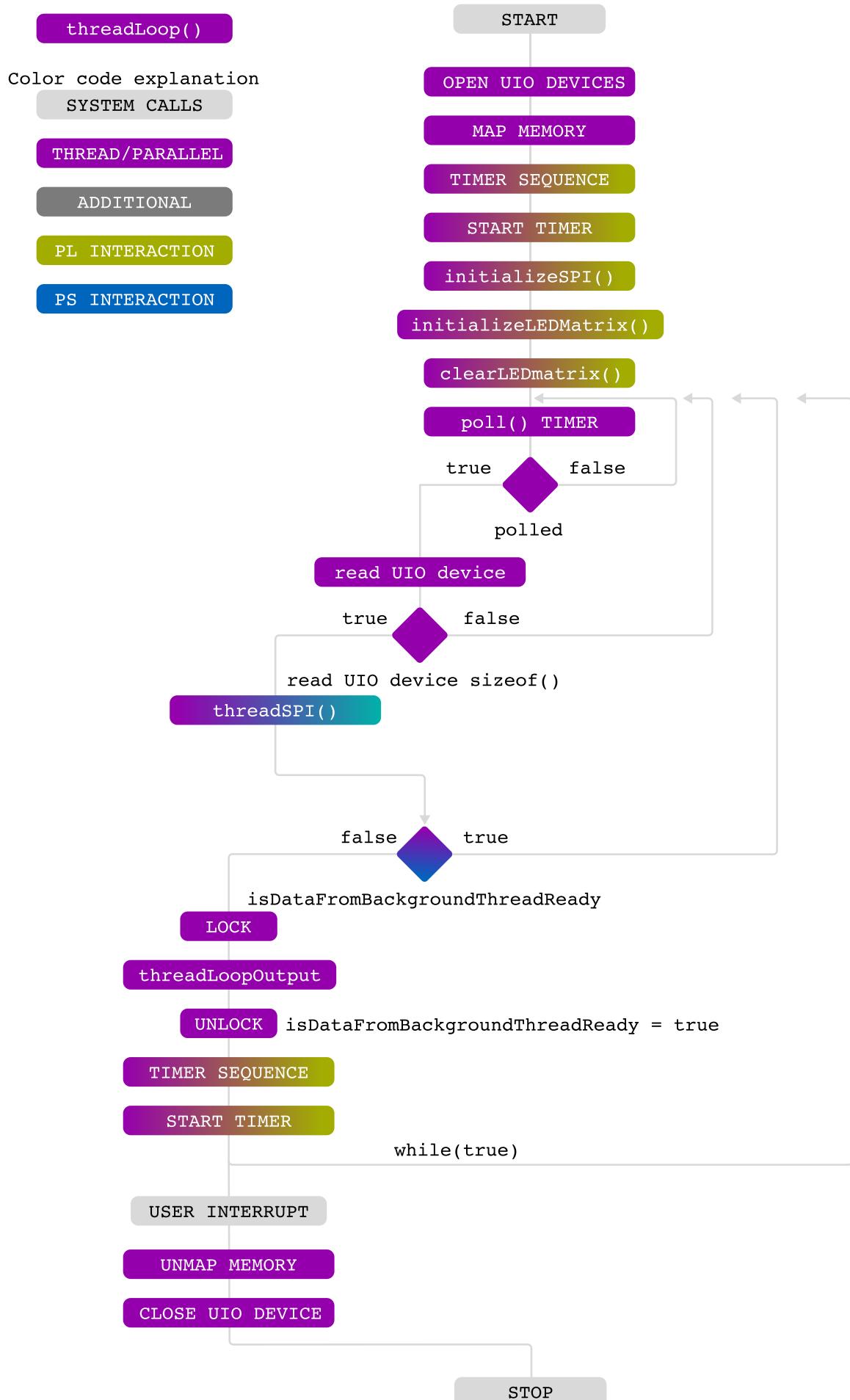
Protože tato aplikace pracuje opět s adresy HW registrů, vytvořených v PL a zobrazitelnými v user

space, je s ohledem na bezpečnost programu vhodné přistupovat k paměti pomocí mapování UIO zařízení do user space pomocí `mmap()` funkce.

Po inicializační části mapování registrů jednotky AXI Timer (postup zobrazen v horní části kódu 17 - 8) a AXI Quad SPI do user space dané aplikace, je v případě ukázkové aplikace využito opět komunikování SPI jednotky s obvodem *MAX7219*. V práci je uvedena kompletní ukázka funkce `threadLoop()`, jejíž algoritmus odráží vývojový diagram 17 - 12. V případě zájmu o nahlédnutí na kódy použitých funkcí

- `initializeSPI(ptrSPI, slaveSelect),`
- `initializeLEDmatrix(ptrSPI, fdSPI, slaveSelect),`
- `clearLEDmatrix(ptrSPI, fdSPI, slaveSelect),`
- `threadSPI(ptrSPI, fdSPI, slaveSelect),`

je možné prozkoumat soubory přílohy této práce. Funkce jsou vloženy do samostatné třídy a jsou zatím ve vývoji. V budoucnu je plánováno vytvořit knihovnu, která umožňuje uživateli komunikovat s bloky jako je AXI Quad SPI a AXI Timer bez znalostí adres jednotlivých registrů.



```

1      /*
2       * @name      threadLoop
3       * @brief     Threaded function for timer and data acquisition.
4       * @todo      Make multiple functions and multiple data acquisitions
5       * parallel but use data only when data from all sources all valid.
6       */
7
8      void threadLoop()
9      {
10         void *timer1ptr;           // pointer to a virtual memory filled by
11         mmap
12         int timer1fd;           // file descriptor of uio to reset
13         interrupt in /proc/interrupts
14         char *uiod;              // name of the uio to reset interrupts
15         uiod = "/dev/uio5";      // check when making changes in a
16         platform
17         int irq_on = 1;           // for writing 0x1 to /dev/uiox
18         uint32_t info = 1;        // in read function of a interrupt
19         checking
20         spiClass spi;
21
22         timer1fd = open(uiod, O_RDWR | O_NONBLOCK); // opening uioX device
23
24         // if error
25         if (timer1fd < 1)
26         {
27             perror("open\n");
28             printf("Invalid UIO device file:%s.\n", uiod);
29         }
30
31         // for polling the interrupt struct
32         struct pollfd fds = {
33             .fd = timer1fd,
34             .events = POLLIN | POLLOUT,
35         };
36
37         // mmap the timer in virtual memory
38         timer1ptr = mmap(NULL, TIMER_MAP_SIZE, PROT_READ | PROT_WRITE,
39         MAP_SHARED, timer1fd, 0);
40
41         // initial values for timer
42         *((unsigned*)(timer1ptr)) = 0X1C0;
43         write(timer1fd, &irq_on, sizeof(irq_on));
44         // *((unsigned*)(timer1ptr + 0x4)) = 0xE8287BFF;
45         // *((unsigned*)(timer1ptr + 0x4)) = 0xFFFFFFF37;
46         *((unsigned*)(timer1ptr + 0x4)) = 0xFFFFF82F; // 10 us
47         // *((unsigned*)(timer1ptr + 0x4)) = 0xFA0A1EFF; // 0.5s
48         *((unsigned*)(timer1ptr)) = 0XE0;

```

```

43
44     // one tick is 0.8 ns, have to wait till data is moved to counter
45     // register
46     // otherwise it wont start, solve maybe later or ask about it
47     std::this_thread::sleep_for(std::chrono::nanoseconds(1));
48
49     *((unsigned *) (timer1ptr + 0x8)) = 0X0;
50     *((unsigned *) (timer1ptr)) = 0XC0;
51
52     off_t slaveSelect = 0x1; // manually selecting slave to make
53     active
54
55     void *ptrSPI; // pointer to a virtual memory filled by mmap
56     int fdSPI; // file descriptor
57     char *uiodSPI; // name what device to open
58     uiodSPI = "/dev/uio4";
59
60     fdSPI = open(uiodSPI, O_RDWR | O_NONBLOCK); // opening device
61
62     // if error
63     if (fdSPI < 1)
64     {
65         perror("open\n");
66         printf("Invalid UIO SPI device file:%s.\n", uiod);
67     }
68     else
69     {
70         std::cout << "Successfully SPI opened device.\n";
71     }
72
73     ptrSPI = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
74     fdSPI, 0);
75     std::this_thread::sleep_for(std::chrono::nanoseconds(1));
76     spi.initializeSPI(ptrSPI, slaveSelect);
77     std::this_thread::sleep_for(std::chrono::nanoseconds(1));
78     spi.initializeLEDmatrix(ptrSPI, fdSPI, slaveSelect);
79     std::this_thread::sleep_for(std::chrono::nanoseconds(1));
80     spi.clearLEDmatrix(ptrSPI, fdSPI, slaveSelect);
81
82     while (true)
83     {
84         while (true) // polling while
85         {
86             int ret = poll(&fds, 1, -1); // poll on a return value
87             // there was change in ret
88             if (ret >= 1)
89             {
90                 ssize_t nb = read(timer1fd, &info, sizeof(info));
91                 if (nb == (ssize_t)sizeof(info))

```

```

88     {
89         /* Do something in response to the interrupt. */
90         printf("Interrupt #%u!\n", info);
91         spi.threadSPI(ptrSPI, fdSPI, slaveSelect);
92         // if timer has finished (interrupt has risen)
93         // copy data / insert data to shared variable
94         if (isDataFromBackgroundThreadReady == false)
95         {
96             gLock.lock();                                // mutex
97             locking - any other thread can't access this variable (think it cannot
98             write or read)
99             threadLoopOutput = threadLoopOutput + 1; // edit the
100            shared variable
101            gLock.unlock();                          // mutex unlock
102            isDataFromBackgroundThreadReady = true; // flag to main
103            while loop that new data is present
104            {
105                }
106                break;
107            }
108        }
109    }
110
111    // resolving and starting timer again
112    *((unsigned *)timer1ptr) = 0X1C0;
113    write(timer1fd, &irq_on, sizeof(irq_on));
114    // *((unsigned *) (timer1ptr + 0x4)) = 0xE8287BFF;
115    // *((unsigned *) (timer1ptr + 0x4)) = 0xFFFFFFF37; // 1 us
116    *((unsigned *) (timer1ptr + 0x4)) = 0xFFFFF82F; // 10 us
117    // *((unsigned *) (timer1ptr + 0x4)) = 0xFA0A1EFF; // 0.5
118    s~*((unsigned *) (timer1ptr)) = 0XE0;
119
120    // one tick is 0.8 ns, have to wait till data is moved to
121    // counter register
122    // otherwise it wont start, solve maybe later or ask about it
123    std::this_thread::sleep_for(std::chrono::nanoseconds(1));
124
125    *((unsigned *) (timer1ptr + 0x8)) = 0X0;
126    *((unsigned *) (timer1ptr)) = 0XC0; // start
127
128    munmap(ptrSPI, MAP_SIZE);
129    close(fdSPI);
130    munmap(timer1ptr, TIMER_MAP_SIZE);
131    close(timer1fd);
132}

```

Kód 17 - 8 threadLoop() funkce, běžící ve vlákně backgroundThread.

17.7 AkcelEROVANé algoritmy (kernely) v PL

V předchozích částech textu byla představena struktura algoritmů aplikací, realizovaných v PS. Algoritmy programů, realizované v PL se odlišují filozofií návrhu jejich struktury. Při tvorbě algoritmů v PS se postupuje dle konvenčních přístupů tvorby C++ programu, kdy jednotlivé příkazy jsou prováděny převážně sekvenčně. Tvorba kernelů je specifická v tom, že sice dochází k využití jazyka C++, ovšem protože je program dále zpracován pomocí HLS, je vhodné se snažit splnit hlavní tři paradigm pro tvorbu algoritmů na FPGA pomocí HLS.

Tyto paradigmata jsou:

- **Producer-Consumer**, jedná se o dekompozici programu tak, aby výsledky jednoho dílčího algoritmu byly využívány pouze jedním dalším dílčím algoritmem, nepoužívat rekurzivní algoritmy apod.,
- **Streaming Data**, využití FIFO pro kontinuální tok dat jedním směrem,
- **Pipelining**, pokud je možné po vykonání dílčích instrukcí na určité sadě dat vykonávat danou instrukci na nové sadě dat, dokud nebyl předchozí algoritmus plně dokončen, dochází ke snížení celkové časové náročnosti programu.

Detailní popis těchto paradigm a jejich ukázkovou realizaci je možné nalézt v oficiální dokumentaci pro HLS od firmy Xilinx, Inc. [57].

V realizovaných algoritmech matematického modelu nejsou dodrženy všechny představené paradigmata. To má za následek snížení kvality překladu pomocí HLS do RTL a následné implementace a tvorby bitstreamu. Toto snížení kvality se projevilo ve větším počtu požadovaných resources a pomalejším překladu. V budoucí práci je vhodné tyto paradigmata lépe implementovat do zavedených algoritmů.

Zdrojové kódy kernelů se nacházejí v souborové příloze této práce.

Filozofií tvorby kernelů bylo rozdělit algoritmus pomocí přístupu *Producer-Consumer* tak, aby nedocházelo k rekurzivním úpravám hodnot proměnných a aby hodnoty daných proměnných byly čteny pouze jednou. Z toho důvodu byly vytvořeny *slice* funkce, které ve smyčce s daným počtem iterací čtou hodnotu z jedné proměnné a vkládají ji do nové proměnné typu pole s pevně daným rozsahem. Hodnota proměnné je vždy čtena z daného indexu pole v algoritmu pouze jednou. Ukázka této funkce pro proměnnou, jež by bez použití *slice* musela být v algoritmu čtena osmkrát, je v kódu 17 - 9.

```
1 void sliceInternalVariables8Parts(float variableIn, float *variableOut)
2 {
3     for(int i = 0; i<8; i++)
4     {
5         variableOut[i] = variableIn;
6     }
7 }
```

Kód 17 - 9 Slice funkce pro kopírování proměnné *variableIn* do pole *variableOut* s osmi polohami.

V první fázi vývoje kernelů bylo používáno knihoven, které autor vytvořil pro simulaci též úlohy pomocí stolního počítače s využitím kompilátoru **gcc** verze **c++14**. I když konstrukce v těchto knihovnách umožňovaly vytvořit přehledné algoritmy a struktury, nebyl tento přístup v práci využit. Čím složitější a více přehlednější popis problému by byl v kernelu realizován, tím problematičtější pro HLS je vytvořit vhodný popis RTL a provést implementaci s tvorbou bitstreamu. Tento problém se opět projeví ve změně použitých resources. Počet potřebných resources může v extrémních případech několikanásobně přesáhnout počet využitých resources v C++ kódě.

nout počet dostupných zdrojů v PL. Tento problém se projevoval převážně u vývojové desky Digilent ZYBO.

17.7.1 Implementace výpočtu diferenciálních rovnic

Výpočet diferenciálních rovnic byl realizován pomocí algoritmu *Runge–Kutta* 4. řádu (RK4). Vhodnost tohoto algoritmu řešení byla ověřena výsledky simulace v MATLAB™u.

Pro zjednodušení algoritmu kernelu byly definovány a deklarovány funkce jednotlivých diferenciálních rovnic. Například pro I - n model byly definovány funkce magnetických toků rotoru $\psi_{2\alpha}$ a $\psi_{2\beta}$. Jednotlivé rovnice soustavy diferenciálních rovnic č. 9 - 14 jsou v kernelu implementovány pomocí kódu 17 - 10. Algoritmus výpočtu časově/hodnotově závislých hodnot je kompletně převzat z obecně známého postupu metody RK4, který využívá iteračního výpočtu soustavy rovnic 17 - 1.

```

1 float psi2alphaFce(float R2MLmDL2, float R2DL2, float i1alpha, float i1beta
, float psi2alpha, float psi2beta, float motorElectricalAngularVelocity)
2 {
3     return ((R2MLmDL2 * i1alpha) - (motorElectricalAngularVelocity *
4         psi2beta) - (R2DL2 * psi2alpha));
5 }
6
6 float psi2betaFce(float R2MLmDL2, float R2DL2, float i1alpha, float i1beta,
7     float psi2alpha, float psi2beta, float motorElectricalAngularVelocity)
8 {
9     return ((R2MLmDL2 * i1beta) + (motorElectricalAngularVelocity *
10        psi2alpha) - (R2DL2 * psi2beta));
11 }
```

Kód 17 - 10 Implementace diferenciálních rovnic č. 9 - 14 do algoritmu kernelu.

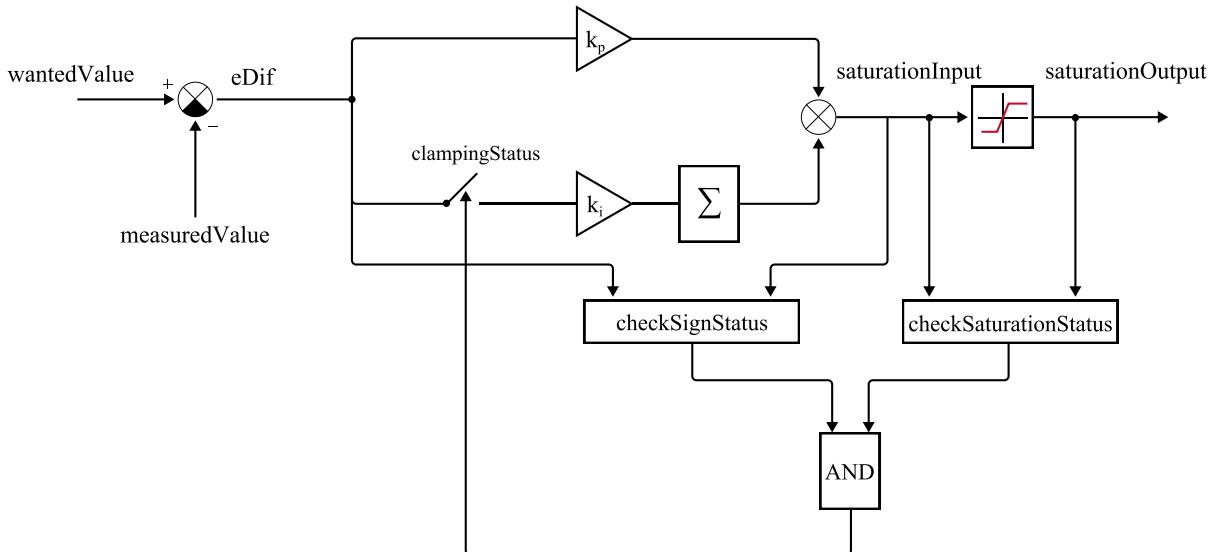
$$\begin{aligned}
 k_{i1} &= h \cdot f(t_n, y_{1n}, y_{2n}, \dots, y_{in}), \\
 k_{i2} &= h \cdot f((t_n + \frac{h}{2}), (y_{1n} + \frac{k_{i1}}{2}), (y_{2n} + \frac{k_{i1}}{2}), \dots, (y_{in} + \frac{k_{i1}}{2})), \\
 k_{i3} &= h \cdot f((t_n + \frac{h}{2}), (y_{1n} + \frac{k_{i2}}{2}), (y_{2n} + \frac{k_{i2}}{2}), \dots, (y_{in} + \frac{k_{i2}}{2})), \\
 k_{i4} &= h \cdot f((t_n + h), (y_{1n} + k_{i3}), (y_{2n} + k_{i3}), \dots, (y_{in} + k_{i3})), \\
 y_{i(n+1)} &= y_{in} + \frac{k_{i1}}{6} + \frac{k_{i2}}{3} + \frac{k_{i3}}{3} + \frac{k_{i4}}{6} + O(h^5),
 \end{aligned} \tag{17 - 1}$$

kde h značí krok metody, index i označuje pořadí proměnné pro kterou je uvedena skladba rovnic.

17.7.2 Implementace regulátorů

V kernelu byly implementovány PI regulátory se strukturou naznačenou na obr. 17 - 13. Pro omezení wind-up efektu byl použit princip *clamping*.

Při realizaci v C++ bylo vhodné realizovat pro regulátory ucelený datový typ struktury, která by obsahovala potřebné proměnné. Tento způsob byl výhodný při implementaci algoritmu v PC, ovšem jak již bylo zmíněno v části *Akcelerované algoritmy (kernely)* v PL, pro překlad pomocí HLS bylo nutné využít datová pole typu float. Tento přístup snížil čitelnost kódu, ale zachoval relativně kvalitní překlad HLS. Blokové schéma principu PI regulátoru, který byl implementován do algoritmu, je zobrazeno v obr. 17 - 13.



Obr. 17 - 13 Blokové schéma regulátoru s ošetřením anti-windup jevu pomocí principu clamping.

17.7.3 Implementace modulace prostorového vektoru

Komparační úrovně v implementaci modulace prostorového vektoru (Space Vector Modulation, SVM) byly získány pomocí *Min-Max* metody. Výpočet komparačních úrovní byl převzat z [58]. Tyto komparační úrovně jsou porovnávány s hodnotami pilovitého průběhu nosného signálu. Na základě porovnaných hodnot jsou nastaveny virtuální stavy **sw1...sw6**, ovládající spínání výkonových polovodičových prvků invertoru. Pilovitý průběh je vytvořen pomocí metody bez využití trigonometrických funkcí, jež by vyžadovaly příliš mnoho resources. Ukázka implementace výpočtu hodnoty pilovitého průběhu v závislosti na čase je v kódu 17 - 11. Ve skutečném kernelu je pro uložení hodnot využito pole typu *float*, ovšem s ohledem na přehlednost není v tomto textu vhodné prezentovat toto provedení.

```

1 float svmCoreClass::generateActualValueTriangleWave(
2     TriangleWaveSettingsType *triangleWaveSettings)
3 {
4     // local variable
5     float triangleActualValue;
6
7     // calculation
8     triangleActualValue = (((4 * triangleWaveSettings->waveAmplitude) /
9         triangleWaveSettings->wavePeriod) * abs(fmod((fmod((triangleWaveSettings
10        ->calculationTime-(triangleWaveSettings->wavePeriod/4)),
11        triangleWaveSettings->wavePeriod) + triangleWaveSettings->wavePeriod),
12        triangleWaveSettings->wavePeriod) - (triangleWaveSettings->wavePeriod
13        /2)) - triangleWaveSettings->waveAmplitude);
14
15     // updating inner wave calculation time based on set initial value
16     // of calculationTime
17     triangleWaveSettings->calculationTime = triangleWaveSettings->
18     calculationTime + triangleWaveSettings->calculationStep;
19
20     // output

```

```

13     return(triangleActualValue);
14 }
```

Kód 17 - 11 Implementace výpočtu aktuální hodnoty pilovitého průběhu. Představená implementace používá struktury typu *TriangleWaveSettingsType*. Ve vytvořeném kernelu je místo struktury použito pole typu *float*.

17.7.4 Implementace modelu invertoru

Model trojfázového můstkového invertoru, napájeného z trojfázového můstkového usměrňovače je implementován v kernelu pomocí soustavy rovnic 17 - 2. [33]

$$\begin{bmatrix} u_{1a} \\ u_{1b} \\ u_{1c} \end{bmatrix} = \frac{1}{3} U_{DC} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} sw1 \\ sw3 \\ sw5 \end{bmatrix}, \quad (17 - 2)$$

kde u_{1a} , u_{1b} , u_{1c} (V) reprezentují velikosti fázového statorového napětí, $sw1$, $sw2$ $sw3$ (1/0) reprezentují stav virtuálních spínačů **sw1**, **sw2**, **sw3** (1/0) v programu.

17.7.5 Implementace modelu asynchronního motoru

Diferenciální rovnice matematického modelu asynchronního motoru (popsané v části *Matematický popis „kompletního“ modelu stroje*) jsou v této práci opět řešeny pomocí metody RK4. Implementace této metody je popsána v *Implementace výpočtu diferenciálních rovnic*.

Oproti krnl_calculateCurVelModel, kernel krnl_calculateInvMot téměř nevyužívá paradigmu *Producer–Consumer*. V budoucí práci je plánováno využít realizovaný systém pro HIL a proto bude třeba *Producer–Consumer* přístup ve značné míře aplikovat. Po aplikaci ovšem dojde k významné komplikaci kódu, protože zavedení *Slice* funkcí způsobí znásobení počtu proměnných v kernelu. Jen na jednu sadu mezivýpočtů prvků k_{i1} pro velikosti proudů statoru $i_{1\alpha}$, $i_{1\beta}$ a magnetických toků rotoru $\psi_{2\alpha}$, $\psi_{2\beta}$ bude místo 6 vstupních proměnných potřeba proměnných 24. (je však možné použít pole a datový typ float, ovšem stále se jedná o zkomplikování kódu)

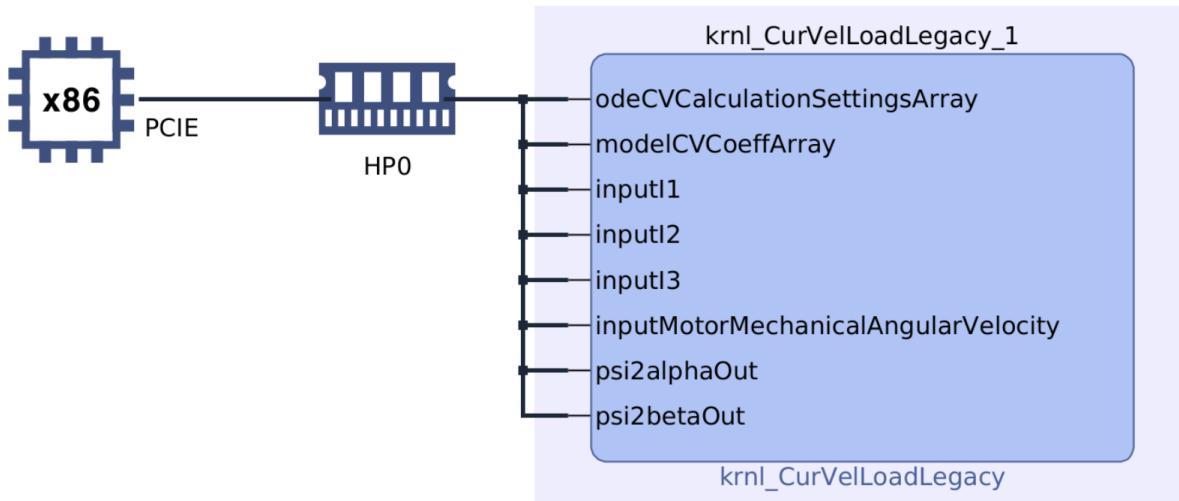
18 Poznatky získané profilováním aplikací

V této sekci budou představeny výsledky a poznatky ohledně vytvořených aplikací, které byly získány profilováním host aplikace (běžící na PS) pomocí knihovny `xrt_coreutil` a programu *Vitis Analyzer*.

18.1 Preloaded Data – Legacy Application

V části *Preloaded Data* byla představena aplikace, která používá starší verzi kódu v kernelu a PS.

Struktura aplikace, která je vizualizována pomocí Vitis Analyzer, je zobrazena na 18 - 1. Informace o skutečném využití zdrojů kernelem je uvedeno v tab. 18 - 3.



Obr. 18 - 1 System diagram – Vitis Analyzer pro Preloaded Data aplikaci.

Tuto aplikaci je vhodné analyzovat z pohledu rychlosti zpracování dat v kernelu v závislosti na velikosti zpracovávaných dat. Aplikace byla testována pro zpracování jednoho milionu a sto tisíc sad hodnot (dále jen jako SH) velikostí fázových statorových proudů ($i_{1a}(t)$, $i_{1b}(t)$, $i_{1c}(t)$), v programu označených jako I₁, I₂, I₃), velikosti mechanické otáčivé rychlosti motoru Ω a složek magnetického toku rotoru $\psi_{2\alpha}$ a $\psi_{2\beta}$.

V tabulce 18 - 1 jsou uvedeny vybrané hodnoty, získané analýzou běhu kernelů.

Dle získaných výsledků je možné konstatovat, že pokud po dokončení výpočtu kernelu dochází k ukládání hodnot do souboru a výpisu hodnot do konzole, není navýšení doby běhu kernelu signifikantní. V případě, že bylo opakováno spuštění aplikace, docházelo k mírnému kolísání změřených hodnot *total runtime*. Tudíž dané navýšení hodnoty mezi variantami ukládání a výpis hodnot (UVH) a bez ukládání a výpisu hodnot (BUVH) je způsobeno nejspíše nestálostí systému a může být předmětem dalšího zkoumání. Zrychlení běhu kernelu se projevuje po opakovaném spuštění kernelu, kdy vlivem zabudované optimalizace nedochází ke změně uložených hodnot, které jsou dány jako konstanty.

Je ovšem vhodné zmínit, že počet zpracovávaných hodnot v kernelu má značný vliv na dobu *total runtime*, přepočtenou na dobu zpracování jedné sady hodnot (SH). Po přepočtu hodnoty *total runtime* na *total runtime / počet SH* je vidět, že při zpracování desetinásobného počtu sad hodnot dochází ke snížení času potřebného na zpracování jedné sady téměř pětkrát. Při přepočtu doby, která je potřebná pro přenos hodnot veličin do a z globální paměti, označené jako *migrateMemObjects*, dochází ke snížení času téměř osmkrát.

Dle uvedených faktů je možné usoudit, že akcelerovaná aplikace pomocí kernelu v PL těží z rozsahu zpracovávaných dat. Pokud je do globální paměti přeneseno velké množství dat, které je kernelem zpracováno, bude doba potřebná na zpracování jedné SH kratší, než pokud bude přeneseno dat méně. Proto se vyplácí akcelerovat aplikace, ve kterých dochází k náročným matematickým výpočtům mnoha hodnot. Výhodné pro akceleraci je, když jsou výpočty prováděny v cyklech.

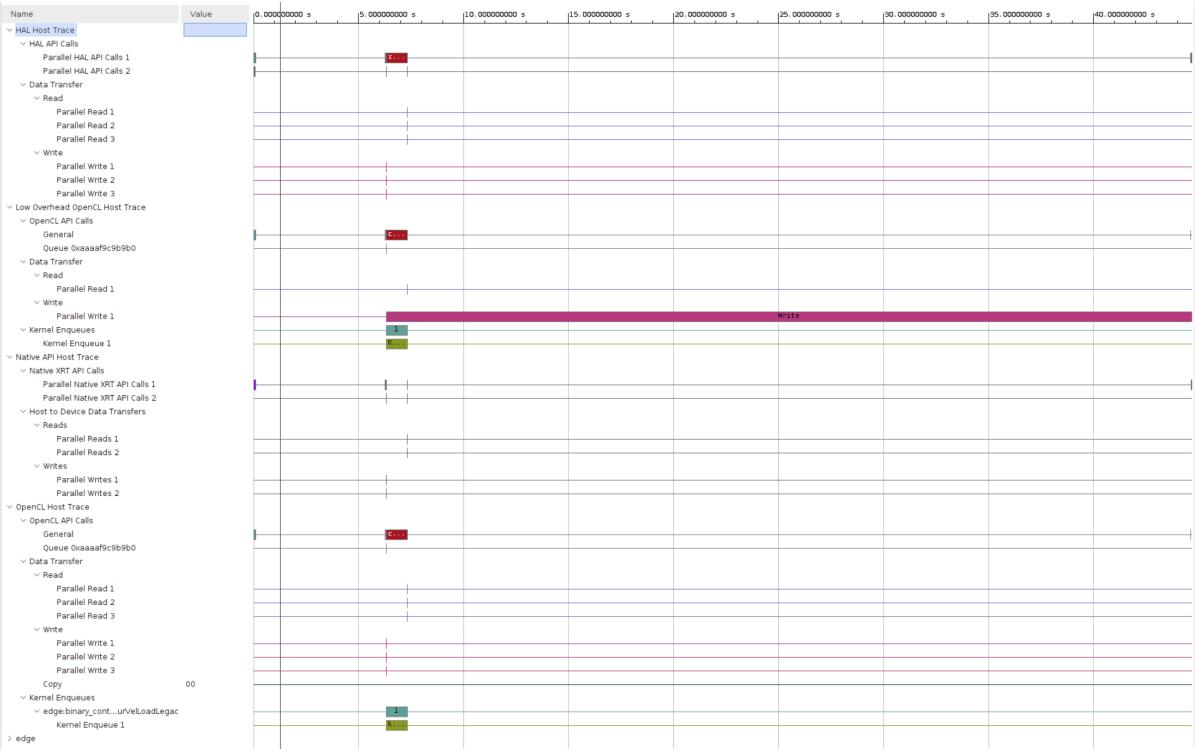
Negativní závěr v případě použití malého množství dat je možné pozorovat i u aplikace *CPU/FPGA*.

Je vhodné doplnit informaci z [5], že Xilinx MPSoC K26 používá pro přenos dat z PS do globální paměti PCIe s propustností dat až 6,0 Gb/s. V tomto případě se nejedná o PCIe pro komunikaci s externími prvky, ale s globální pamětí umístěné na MPSoC. Tudíž samotný přesun dat mezi pamětí PS a globální pamětí by měl probíhat vysokou rychlostí. V jednom z testovaných průběhů aplikace *Preloaded Data* dochází k blokování běhu programu vlivem zápisu dat do globální paměti z PS po dobu přibližně 1,20 ms a vlivem čtení výsledných hodnot z této paměti po dobu přibližně 0,38 ms.

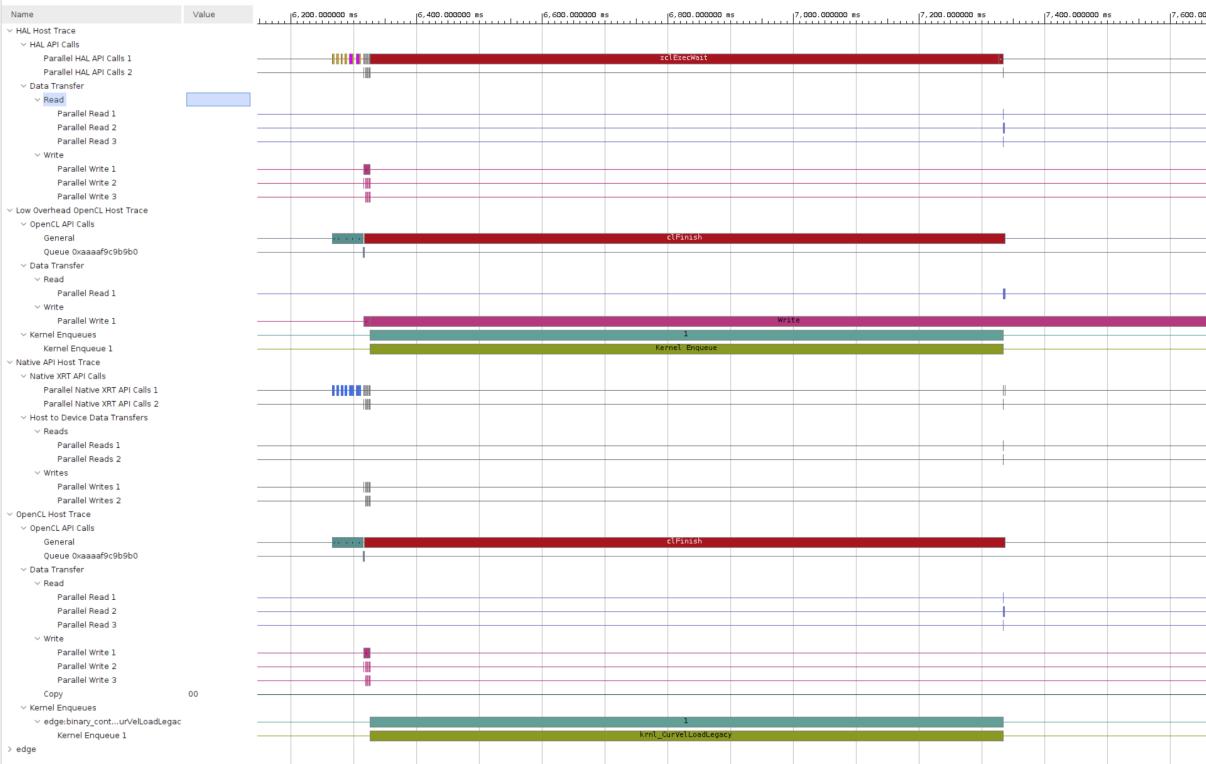
Je vhodné upozornit, že sice v obr. 18 - 3 je po přiblížení fialově naznačená aktivita *Paralel Write* nazývána jako *Host to Memory*, ale v okamžiku běhu kernelu by nemělo docházet host programem k zápisu dat do globální paměti. Jedná se tedy nejspíše o zápis PL do globální paměti, ze které budou poté pomocí jiné instrukce čteny výsledné hodnoty do paměti PS. Ale i tato úvaha může být nesprávná, protože po ukončení běhu kernelu a vykonání instrukce *Read*, je až do konce celkového běhu programu vykonávána instrukce *Paralel Write*.

Tab. 18 - 1 Porovnání vybraných hodnot běhu kernelu v aplikace Preloaded Data – Legacy App pro 1 milion a 100 tisíc sad hodnot (SH). (UVH – ukládání a výpis hodnot, BUVH – bez ukládání a výpisu hodnot)

Preloaded Data					
název	krok	total runtime (ms)	runtime 1 SH (μ s)	migrateMemObjects (ms)	clFinish (ms)
100 k SH, UVH	$1 \cdot 10^{-5}$	502,284	5,02284	0,749	503,691
1 M SH, UVH	$1 \cdot 10^{-6}$	1007,880	1,00788	0,940	1019,280
1 M SH, BUVH	$1 \cdot 10^{-6}$	1005,070	1,00507	0,907	1016,220



Obr. 18 - 2 Timeline Trace – Vitis Analyzer pro Preloaded Data aplikaci při zpracování 1 M SH. Na obrázku je zobrazen celý životní cyklus aplikace od startu až po ukončení.



Obr. 18 - 3 Timeline Trace – Vitis Analyzer pro Preloaded Data aplikaci při zpracování 1 M SH. Na obrázku je zobrazena část, kdy je spuštěn kernel a aplikace v PS čeká na výsledky jeho kalkulací.

18.2 CPU/FPGA

Aplikaci *CPU/FPGA* nebylo možné v její původní verzi analyzovat. V případě, že byla aplikace spuštěna pro 1 M SH (1 M souborů hodnot), aplikace běžela, ale nedokončila se v definovaném čase a byla *PetaLinux* systémem automaticky ukončena. Tato situace vzniká nejspíše vlivem toho, že systém není schopen při takovém množství iterací spuštění kernelu ukládat soubory a hodnoty analýzy. V některých případech byla systémem zobrazena hláška *Killed*. V původním systému, *PetaLinux* bez aplikovaného RT patche, aplikace nebyla nikdy dokončena akceptovatelným způsobem. Z udaných důvodů vyplývá, že je vhodné analyzovat tuto aplikaci na menší SH. Analýza byla provedena pro 1000 SH a 100 SH.

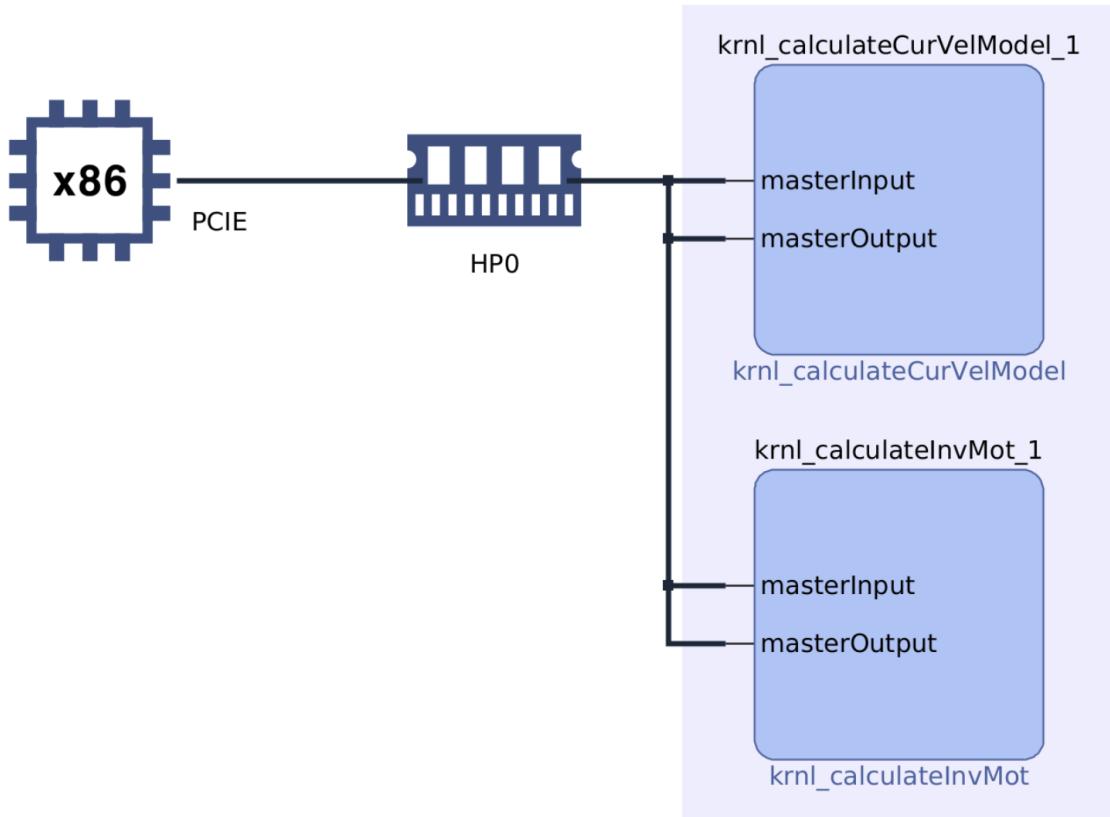
V tabulce 18 - 2 jsou uvedeny výsledky daných běhů programu. Protože je aplikace postavena na iteracích spuštění kernelu a nikoliv na iteracích algoritmu uvnitř kernelu, je možné pozorovat odlišené výsledky, než které byly zjištěny u aplikace *Preloaded Data – Legacy Application*. V aplikaci *CPU/FPGA* nedochází při zvýšení počtu SH ke snížení doby potřebné na jednu iteraci běhu kernelu.

Ve stávající architektuře aplikace dochází k přesunu dat mezi `krnl_calculateCurVelModel` a `krnl_calculateInvMot` pomocí PS. Architektura kernelů a PS je zobrazena na obr. 18 - 4. Tento přenos dat představuje další instrukce, které je nutné vykonávat a může způsobit zpoždění běhu aplikace. Z toho důvodu je vhodné v nadcházejících pracích analyzovat architekturu, kdy dochází k přenosu dat mezi jednotlivými kernely napřímo pomocí streamingu dat. Tento způsob přenosu dat je představen firmou Xilinx v [59].

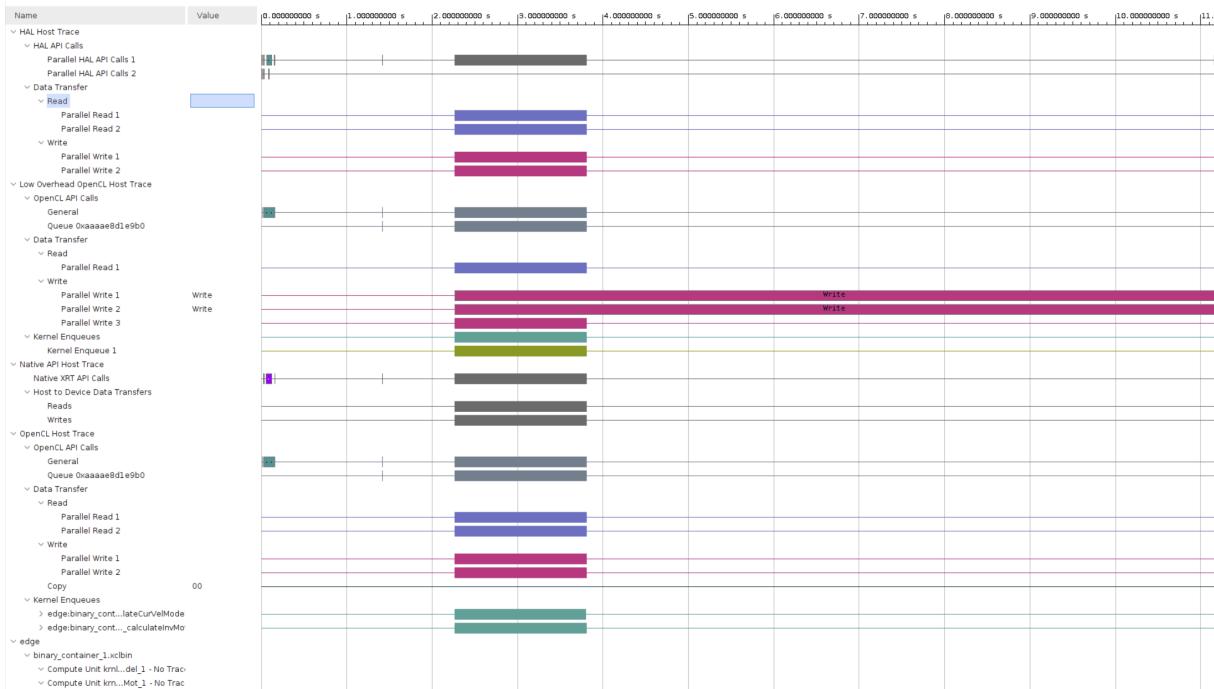
Doby, potřebné na výpočet jedné SH, které je možné vyčíst z tabulky 18 - 2, a doby, které byly získány pomocí profilování aplikace naznačují, že tento způsob tvorby akcelerovaných aplikací není vhodný pro řízení elektrických pohonů nebo pro realizaci HIL. Vhodnějším přístupem, který je vhodné zvolit a prozkoumat, je využití programu Vitis HLS pro tvorbu IP, které budou integrovány do PL v programu Vivado. Realizované algoritmy v IP budou nezávislé na PS a budou se chovat, jako kdyby byly realizovány v samostatném FPGA. Tento způsob realizace by měl splňovat požadavky na dobu zpracování požadovaných SH, která bývá v některých aplikacích menší než $5 \mu\text{s}$.

Tab. 18 - 2 Vybrané hodnoty běhu kernelu v podaplikaci CPU/FPGA.

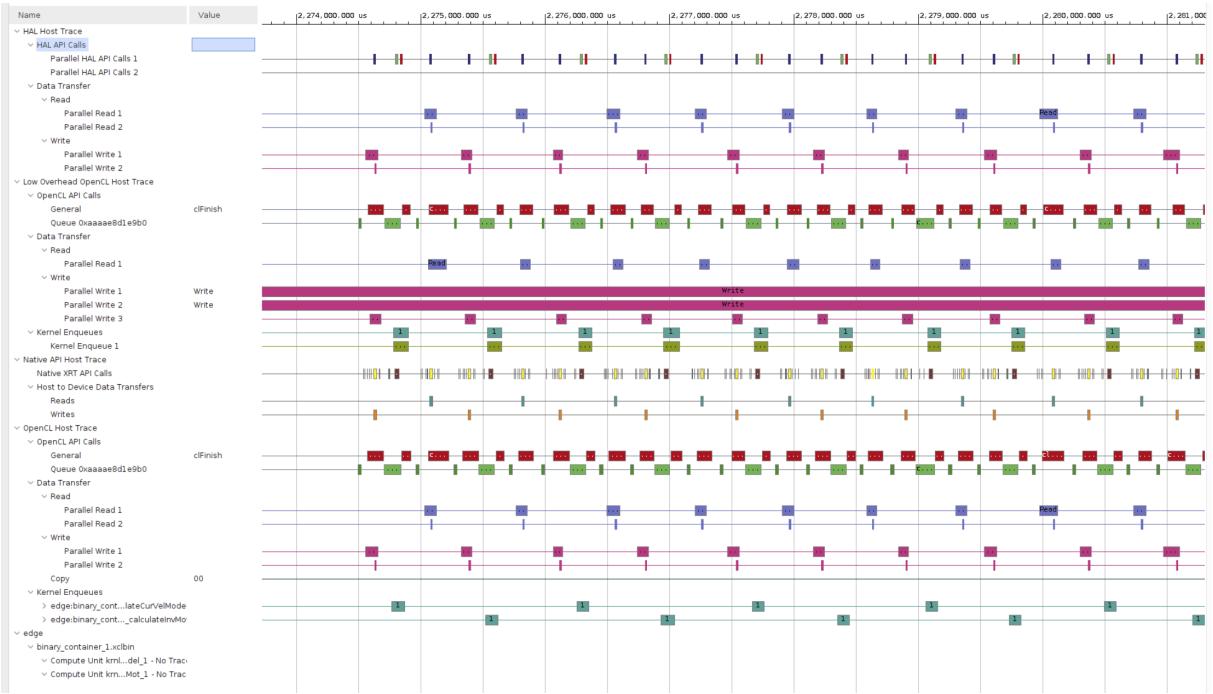
CPU/FPGA		
název	hodnota (ms)	hodnota 1 SH (ms)
krok	$1 \cdot 10^{-3}$	x
1000 SH		
total runtime krnl_calculateCurVelModel	110,916	0,110916
total runtime krnl_calculateInvMot	110,486	0,110486
device execution time	1539,100	1,53910
migrateMemObjects	115,889	0,11589
clFinish	626,687	0,626687
100 SH		
total runtime krnl_calculateCurVelModel	11,552	0,11552
total runtime krnl_calculateInvMot	10,442	0,10442
device execution time	141,249	1,41249
migrateMemObjects	10,678	0,10678
clFinish	59,294	0,59294



Obr. 18 - 4 System diagram – Vitis Analyzer pro CPU/FPGA aplikaci.



Obr. 18 - 5 Timeline Trace – Vitis Analyzer pro CPU/FPGA aplikaci při zpracování 1000 SH. Na obrázku je zobrazen celý životní cyklus aplikace od startu až po ukončení.



Obr. 18 - 6 Timeline Trace – Vitis Analyzer pro CPU/FPGA aplikaci při zpracování 1000 SH hodnot. Na obrázku je zobrazena přiblížená část, kdy dochází k iteraci spuštění jednotlivých kernelů.

18.3 Využití zdrojů pro PL jednotlivých akcelerovaných aplikací

V tabulce 18 - 3 je prezentováno využití zdrojů LUT, registrů, BRAM, URAM a DSP jednotlivými akcelerovanými aplikacemi (kernely). Je vidět, že kernel krnl_calculateInvMot by s vysokou pravděpodobností mohl být realizován i ve vývojové desce Digilent Zybo.

Naopak krnl_calculateCurVelModel, který obsahuje *I-n* model simulovaného motoru, regulátory a SVM, vyžaduje v aktuálním provedení množství zdrojů, které prezentovaná deska Digilent Zybo není schopna poskytnout.

Tab. 18 - 3 Využití zdrojů PL pro akcelerované aplikace.

Kernel	LUT	Registry	BRAM	URAM	DSP
krnl_CurVelLoadLegacy	6 520	8 003	2	0	19
krnl_calculateCurVelModel	23 713	23 490	3	0	103
krnl_calculateInvMot	14 207	16 319	9	0	75

Závěr

Práce je zaměřená na možné využití platforem SoC, zejména heterogenních s FPGA, v oboru řízení elektrických pohonů. V úvodních kapitolách se autor věnoval rozdílu SOM a SoC struktur. Protože je při řízení elektrických pohonů a nebo analýze jejich chování potřeba analyzovat velké množství dat v reálném čase, analyzoval autor struktury, které obsahují i logická programovatelná pole, jež jsou schopny náročné výpočty provádět. Pro demonstraci využití byly vybrány a porovnány dvě vývojové desky, jež využívají SoC a SOM od firmy Xilinx, Inc. Prvotní vývoj akcelerované aplikace probíhal s využitím desky Digilent Zynq-7000, ovšem z důvodu nedostatečného množství zdrojů v PL bylo její využití pro realizaci ukázkové aplikace zavrhnuto. Realizace ukázkové aplikace byla nakonec prováděna s využitím Kria KR260 s K26 SOM, jež obsahuje dostatečné množství zdrojů pro vytvoření akcelerované simulace řízení asynchronního motoru pomocí FOC.

V práci je představen postup, kterým je možné vytvořit HW design, využitý při tvorbě *PetaLinux* systému. Aplikace na vývojové desky bylo možné realizovat jako *Bare Metal* nebo systémové, s využitím *PetaLinux* systému, který je spuštěn na PS.

Realizované algoritmy simulace jsou popsány vývojovými diagramy, které nastiňují způsob práce s akcelerovanými aplikacemi.

V závěru této práce byla provedena analýza použitého přístupu k akcelerovaným aplikacím a bylo zjištěno, že zvolený způsob realizace není vhodný. Pro řízení elektrických pohonů v reálném čase není vhodné realizovat část aplikace v PS a část aplikace v PL. Přenos dat do globální paměti z PS do PL a zpět spolu se zpožděním spouštěním kernelu vnáší do algoritmu příliš velké časové konstanty, které při řízení v reálném čase není možné tolerovat.

Autor v rámci analýzy realizovaných aplikací nastiňuje další způsoby tvorby aplikace, kterými by bylo tyto časové konstanty možné minimalizovat. Pro zrychlení komunikace mezi kernely a PS by bylo možné realizovat vyšší míru paralelizace algoritmů, zkombinovanou s přístupem streamingu dat do volně běžícího (free running) kernelu. Další možností je realizovat algoritmy do jednotlivých IP bloků pomocí Vitis HLS, nikoliv Vitis IDE, a tyto bloky realizovat přímo ve struktuře PL s pomocí SW Vivado. Tento přístup by znamenal, že prakticky veškeré algoritmy by byly realizované v FPGA a PS aplikace by sloužila pouze ke konfiguraci hodnot v daných HW registrech.

Tato diplomová práce vytváří základy pro možnou navazující práci, ve které bude vyhledáván způsob, jakým realizovat řízení v reálném čase s pomocí moderních heterogenních systémů se SoC a SOM. Další práce by měla být zaměřena na dodržení potřebných časových a bezpečnostních opatření aplikace pro produkční prostředí. Kromě řízení v reálném čase je možné v navazující práci řešit využití těchto platform pro realizaci HIL systémů. Realizované systémy HIL by díky těmto platformám mohly být oproti stávajícím komerčním řešením méně energeticky a finančně náročné.

Literatura

- [1] HARDING, Sharon. What Is an SoC? A Basic Definition. In: *Blog post* [online]. 11. 09. 2019 [cit. 2023-03-18]. Dostupné z: <https://www.tomshardware.com/reviews/glossary-soc-system-on-chip-definition,5890.html>.
- [2] XILINX, Inc. System-on-Modules (SOMs): How and Why to Use Them. In: *Xilinx Website* [online]. [B.r.] [cit. 2023-03-10]. Dostupné z: <https://www.xilinx.com/products/som/what-is-a-som.html>.
- [3] XILINX, Inc. Kria KR260 Robotics Starter Kit. In: *Xilinx Website* [online]. [B.r.] [cit. 2023-03-10]. Dostupné z: <https://www.xilinx.com/products/som/kria/kr260-robotics-starter-kit.html>.
- [4] XILINX, Inc. Kria SOM Carrier Card Design Guide (UG1091). In: *AMD Xilinx Documentation Portal* [online]. 27. 07. 2022 [cit. 2023-03-18]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug1091-carrier-card-design/Introduction>.
- [5] XILINX, Inc. Kria K26 SOM Data Sheet (DS987). In: *AMD Xilinx Documentation Portal* [online]. 26. 07. 2022 [cit. 2023-03-18]. Dostupné z: <https://docs.xilinx.com/r/en-US/ds987-k26-som>.
- [6] XILINX, Inc. Antmicro's open source Kria K26 Devboard. In: *GitHub* [online]. [B.r.] [cit. 2023-03-10]. Dostupné z: <https://github.com/antmicro/kria-k26-devboard>.
- [7] SASS, Ronald; SCHMIDT, Andrew G. *Embedded systems design with platform FPGAs: principles and practices*. Boston: Morgan Kaufmann, 2010. ISBN 0123743338.
- [8] ANDINA, Juan J. R.; TORRE ARNANZ, Eduardo de la; VALDÉS PEÑA, María D. *FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics*. 1. vyd. Bosa Roca: CRC Press, 2017;2015; ISBN 9781439896990;1439896992;
- [9] What Is Accelerated Computing, and Why Is It Important? In: *Xilinx, Inc.* [Online]. [B.r.] [cit. 2022-11-03]. Dostupné z: <https://www.xilinx.com/applications/adaptive-computing/what-is-accelerated-computing-and-why-is-it-important.html>.
- [10] NALCACI, Gamze; YILDIRIM, Doğan; CADIRCI, Isik; ERMIS, Muammer. Selective Harmonic Elimination for Variable Frequency Traction Motor Drives Using Harris Hawks Optimization. *IEEE Transactions on Industry Applications*. 2022, roč. 58, č. 4, s. 4778–4791. Dostupné z DOI: 10.1109/TIA.2022.3174828.
- [11] APPLE, Inc. Explore the new system architecture of Apple silicon Macs. In: *Apple WWDC Video* [online]. [B.r.] [cit. 2023-03-18]. Dostupné z: <https://developer.apple.com/videos/play/wwdc2020/10686/>.
- [12] PANG, Aiken; MEMBREY, Peter; SLUŽBA), SpringerLink (online. *Beginning FPGA: Programming Metal: Your brain on hardware: Programming Metal*. Berkeley, CA: Apress, 2017;2016; č. Book, Whole. Dostupné také z: <https://go.exlibris.link/JzysQtpz>.
- [13] Sphery vs. shapes, the first raytraced game that is not software. In: *GitHub* [online]. 14. 07. 2022 [cit. 2022-11-06]. Dostupné z: <https://github.com/JulianKemmerer/PipelineC-Graphics/blob/main/doc/Sphery-vs-Shapes.pdf>.
- [14] Ray tracing (graphics). In: *Wikipedia* [online]. 06. 11. 2022 [cit. 2022-11-06]. Dostupné z: [https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics)).

- [15] GROVER, Naresh; K.SONI, M. Reduction of Power Consumption in FPGAs - An Overview. *International journal of information engineering and electronic business*. 2012, roč. 4, č. 5, s. 50–69. ISBN 2074-9023.
- [16] Amazon EC2 F1 Instances. In: *Amazon AWS* [online]. [B.r.] [cit. 2022-11-06]. Dostupné z: <https://aws.amazon.com/ec2/instance-types/f1/>.
- [17] VANDERBAUWHEDE, Wim; BENKRID, Khaled. *High-Performance Computing Using FPGAs*. Springer Publishing Company, Incorporated, 2013. ISBN 1461417902.
- [18] RODRÍGUEZ-ANDINA, Juan J.; VALDÉS-PEÑA, María D.; MOURE, María J. Advanced Features and Industrial Applications of FPGAs---A Review. *IEEE Transactions on Industrial Informatics*. 2015, roč. 11, č. 4, s. 853–864. Dostupné z DOI: 10.1109/TII.2015.2431223.
- [19] Hardware-in-the-Loop (HIL) Simulation. In: *The MathWorks, Inc.* [Online]. [B.r.] [cit. 2022-11-06]. Dostupné z: <https://www.mathworks.com/discovery/hardware-in-the-loop-hil.html>.
- [20] NAOUAR, Mohamed-Wissem; MONMASSON, Eric; NAASSANI, Ahmad Ammar; SLAMA-BELKHODJA, Ilhem; PATIN, Nicolas. FPGA-Based Current Controllers for AC Machine Drives---A Review. *IEEE Transactions on Industrial Electronics*. 2007, roč. 54, č. 4, s. 1907–1925. Dostupné z DOI: 10.1109/TIE.2007.898302.
- [21] DIGILENT, Inc. Zybo. In: *Digilent Documentation* [online]. [B.r.] [cit. 2022-11-11]. Dostupné z: <https://digilent.com/reference/programmable-logic/zybo/start>.
- [22] DIGILENT, Inc. Zybo Z7 Migration Guide. In: *Digilent Documentation* [online]. [B.r.] [cit. 2022-11-11]. Dostupné z: <https://digilent.com/reference/programmable-logic/zybo-z7/migration-guide>.
- [23] XILINX, Inc. Zynq-7000 SoC Technical Reference Manual. In: *Xilinx Documentation* [online]. 02. 04. 2021 [cit. 2022-11-11]. Dostupné z: <https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM>.
- [24] DIGILENT, Inc. Zybo Reference Manual. In: *Digilent Documentation* [online]. [B.r.] [cit. 2022-11-11]. Dostupné z: <https://digilent.com/reference/programmable-logic/zybo/reference-manual>.
- [25] XILINX, Inc. Kria KR260 Robotics Starter Kit User Guide (UG1092). In: *AMD Xilinx Documentation Portal* [online]. 17. 05. 2022 [cit. 2023-04-05]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug1092-kr260-starter-kit/Interfaces>.
- [26] XILINX, Inc. Kria K26 System-on-Module. In: *AMD Xilinx Product Brief* [online]. [B.r.] [cit. 2023-04-05]. Dostupné z: <https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-k26-product-brief.pdf>.
- [27] XILINX, Inc. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). In: *AMD Xilinx Documentation Portal* [online]. [B.r.] [cit. 2022-11-18]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/>.
- [28] XILINX, Inc. XTP743 - Kria KR260 Starter Kit Carrier Card Schematics (v1.0). In: *AMD Xilinx Board Files* [online]. 09. 06. 2022 [cit. 2023-04-06]. Dostupné z: <https://www.xilinx.com/member/forms/download/design-license.html?cid=bad0ada6-9a32-427e-a793-c68fed567427&filename=xtp743-kr260-schematic.zip>.

- [29] XILINX, Inc. XTP685 - Kria K26 SOM XDC File (v1.0). In: *AMD Xilinx Board Files* [online]. 14. 05. 2021 [cit. 2023-04-06]. Dostupné z: <https://www.xilinx.com/member/forms/download/design-license.html?cid=29e0261a-9532-4a47-bb06-38c83bbbb8c0&filename=xtp685-kria-k26-som-xdc.zip>.
- [30] ADMIN, Confluence Wiki; ROY, Debraj; DYLAN. Embedded SW Support. In: *Xilinx Wiki* [online]. 28. 02. 2023 [cit. 2023-04-06]. Dostupné z: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841631/Embedded+SW+Support>.
- [31] XILINX, Inc. Pre-Built Applications for Kria System-on-Modules. In: *Xilinx Website* [online]. [B.r.] [cit. 2023-04-06]. Dostupné z: <https://www.xilinx.com/products/app-store/kria.html>.
- [32] FOUNDATION, Linux. Real-Time Linux. In: *Linux Foundation DokuWiki* [online]. [B.r.] [cit. 2023-04-06]. Dostupné z: <https://wiki.linuxfoundation.org/realtime/start>.
- [33] LIPČÁK, Ondřej; BAUER, Jan. Doprovodný materiál k přednáškám. In: *Materiál k přednáškám předmětu BIMI4EPT* [online]. [B.r.] [cit. 2023-02-28]. Dostupné z: <https://moodle.cvut.cz>.
- [34] KOBRLE, Pavel; PAVELKA, Jiří. *Elektrické pohony a jejich řízení*. 3. přepracované vydání. V Praze: České vysoké učení technické, 2016. ISBN 978-80-01-06007-0.
- [35] XILINX, Inc. Downloads. In: *AMD Xilinx Downloads* [online]. [B.r.] [cit. 2022-11-19]. Dostupné z: <https://www.xilinx.com/support/download.html>.
- [36] XILINX, Inc. Downloads. In: *AMD Xilinx PetaLinux Tools* [online]. [B.r.] [cit. 2022-11-19]. Dostupné z: <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html>.
- [37] XILINX, Inc. PetaLinux Tools Documentation: Reference Guide (UG1144). In: *AMD Xilinx Documentation Portal* [online]. [B.r.] [cit. 2022-11-18]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide>.
- [38] REGHENZANI, Federico; MASSARI, Giuseppe; FORNACIARI, William. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.* 02/2019, roč. 52, č. 1. ISSN 0360-0300. Dostupné z DOI: 10.1145/3297714.
- [39] COMPANY], LogicTronix [FPGA Design + Machine Learning. Real Time Optimization in Petalinux with RT Patch on MPSoC. In: *Hackster.io, an Avnet Community* [online]. 10. 12. 2021 [cit. 2023-04-07]. Dostupné z: <https://www.hackster.io/LogicTronix/real-time-optimization-in-petalinux-with-rt-patch-on-mpsoc-5f4832>.
- [40] ERRAPART, Andrei; IGELBRINK, Felix. How to install the linux-rt (Real-Time) patch. In: *Trenz Electronic Wiki* [online]. 05. 10. 2016 [cit. 2023-04-07]. Dostupné z: <https://wiki.trenz-electronic.de/display/PD/How+to+install+the+linux-rt+%28Real-Time%29+patch>.
- [41] XILINX, Inc. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973). In: *AMD Xilinx Documentation Portal* [online]. [B.r.] [cit. 2022-11-18]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug973-vivado-release-notes-install-license/>.
- [42] DIGILENT, Inc. Vivado Board Files for Digilent FPGA Boards, 2022. In: *GitHub.io* [online]. 25. 03. 2022 [cit. 2022-11-26]. Dostupné z: <https://github.com/Digilent/vivado-boards>.
- [43] DIGILENT, Inc. Installing Vivado, Vitis, and Digilent Board Files. In: *Digilent Reference* [online]. [B.r.] [cit. 2022-11-26]. Dostupné z: <https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis>.

- [44] KNITTER, Whitney. Getting Started with the Kria KR260 in PetaLinux 2022.1. In: *Hackster.io, an Avnet Community* [online]. 12. 08. 2022 [cit. 2023-04-10]. Dostupné z: <https://www.hackster.io/whitney-knitter/getting-started-with-the-kria-kr260-in-petalinux-2022-1-daec16>.
- [45] KNITTER, Whitney. Add Peripheral Support to Kria KR260 Vivado 2022.1 Project. In: *Hackster.io, an Avnet Community* [online]. 11. 08. 2022 [cit. 2023-04-10]. Dostupné z: <https://www.hackster.io/whitney-knitter/add-peripheral-support-to-kria-kr260-vivado-2022-1-project-874960>.
- [46] KNITTER, Whitney. Getting Started with the Kria KR260 in Vivado 2022.1. In: *Hackster.io, an Avnet Community* [online]. 31. 07. 2022 [cit. 2023-04-10]. Dostupné z: <https://www.hackster.io/whitney-knitter/getting-started-with-the-kria-kr260-in-vivado-2022-1-33746d>.
- [47] XILINX, Inc. Vitis Custom Embedded Platform Creation Example on KV260 – Step 1: Create the Vivado Hardware Design and Generate XSA. In: *Xilinx Vitis Tutorials Github Repository* [online]. 22. 05. 2022 [cit. 2023-04-13]. Dostupné z: https://xilinx.github.io/Vitis-Tutorials/2022-1/build/html/docs/Vitis_Platform_Creation/Design_Tutorials/01-Edge-KV260/step1.html.
- [48] XILINX, Inc. Zynq UltraScale+ MPSoC Processing System Product Guide (PG201). In: *Xilinx Documentation* [online]. 11. 05. 2021 [cit. 2023-04-13]. Dostupné z: <https://docs.xilinx.com/r/en-US/pg201-zynq-ultrascale-plus-processing-system/Fabric-Reset-Enable>.
- [49] ADMIN, Confluence Wiki; BHATT, Madhav. Zynq UltraScale+ MPSoC Restart solution. In: *Xilinx Wiki* [online]. 24. 08. 2022 [cit. 2023-04-13]. Dostupné z: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842142/GPIO+User+Space+App>.
- [50] ZAKOPAL, Petr et al. [Kria SOM KR260 Starter Kit] Schematic (pdf) vs constrains (xdc) pin confusion. Possible explanation on fan pinout. In: *Xilinx Support Community Forum* [online]. 18. 03. 2023 [cit. 2023-04-13]. Dostupné z: https://support.xilinx.com/s/question/0D54U00006alUwcSAE/kria-som-kr260-starter-kit-schematic-pdf-vs-constrains-xdc-pin-confusion-possible-explanation-on-fan-pinout?language=en_US.
- [51] XILINX, Inc. Vitis Custom Embedded Platform Creation Example on KV260 – Step 2: Step 2: Create the Software Components. In: *Xilinx Vitis Tutorials Github Repository* [online]. 16. 05. 2022 [cit. 2023-04-14]. Dostupné z: https://xilinx.github.io/Vitis-Tutorials/2022-1/build/html/docs/Vitis_Platform_Creation/Design_Tutorials/01-Edge-KV260/step2.html.
- [52] XILINX, Inc. AXI Quad SPI v3.2 LogiCORE IP Product Guide. In: *AMD Xilinx Documentation Portal* [online]. 26. 04. 2022 [cit. 2023-04-14]. Dostupné z: <https://docs.xilinx.com/r/en-US/pg153-axi-quad-spi>.
- [53] XILINX, Inc. AXI Timer v2.0 LogiCORE IP Product Guide. In: *AMD Xilinx Documentation Portal* [online]. 05. 10. 2016 [cit. 2023-04-14]. Dostupné z: <https://docs.xilinx.com/v/u/en-US/pg079-axi-timer>.
- [54] LIMITED, Linaro. The DeviceTree Specification. In: *Specification page* [online]. [B.r.] [cit. 2023-04-14]. Dostupné z: <https://www.devicetree.org/>.
- [55] Device Tree for Dummies! - Thomas Petazzoni, Free Electrons. In: *YouTube* [online]. 15. 11. 2013 [cit. 2023-04-14]. Dostupné z: https://youtu.be/m_NyYEBxfn8. Kanál uživatele The Linux Foundation.
- [56] XILINX, Inc. DFX-MGR Github Repository. In: *GitHub* [online]. 23. 08. 2022 [cit. 2023-04-15]. Dostupné z: <https://github.com/Xilinx/dfx-mgr>.

- [57] XILINX, Inc. Vitis High-Level Synthesis User Guide (UG1399). In: *AMD Xilinx Documentation Portal* [online]. 07. 12. 2022 [cit. 2023-04-20]. Dostupné z: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls>.
- [58] CORPORATION, Microsemi. Space Vector Modulation v4.1, User Guide, UG0468. In: *User Guide* [online]. [B.r.] [cit. 2023-04-20]. Dostupné z: https://www.microsemi.com/document-portal/doc_download/133496-ug0468-space-vector-modulation-v4-1-user-guide.
- [59] XILINX, Inc. Stream Free Running Kernel (HLS C/C++). In: *Xilinx Vitis Accel Examples Github Repository* [online]. [B.r.] [cit. 2023-04-24]. Dostupné z: https://xilinx.github.io/Vitis_Accel_Examples/2022.2/html/streaming_free_running_k2k.html.
- [60] HOSSEINABADY, Mohammad. Vitis 2021.1 Embedded Platform for Zybo-Z7-20, 2018. In: *Hackster.io, an Avnet Community* [online]. 16. 08. 2021 [cit. 2022-11-26]. Dostupné z: <https://www.hackster.io/mohammad-hosseinabady2/vitis-2021-1-embedded-platform-for-zybo-z7-20-d39e1a>.
- [61] XILINX, Inc. SoCs with Hardware and Software Programmability. In: *Xilinx Website* [online]. [B.r.] [cit. 2022-11-11]. Dostupné z: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.
- [62] DIGILENT, Inc. Zybo Petalinux BSP Project, 2018. In: *GitHub.io* [online]. 29. 03. 2018 [cit. 2022-11-26]. Dostupné z: <https://github.com/Digilent/Petalinux-Zybo>.
- [63] MESSINGER, Roy. GPIO and Petalinux - Part 1. In: *LinkedIn Article* [online]. 20. 06. 2020 [cit. 2023-02-28]. Dostupné z: <https://www.linkedin.com/pulse/gpio-petalinux-part-1-roy-messinger/>.
- [64] MESSINGER, Roy. GPIO and Petalinux - Part 2. In: *LinkedIn Article* [online]. 27. 06. 2020 [cit. 2023-02-28]. Dostupné z: <https://www.linkedin.com/pulse/gpio-petalinux-part-2-roy-messinger/>.
- [65] MESSINGER, Roy. GPIO and Petalinux - Part 3 (Go, UIO, Go!) In: *LinkedIn Article* [online]. 23. 07. 2021 [cit. 2023-02-28]. Dostupné z: <https://www.linkedin.com/pulse/gpio-petalinux-part-3-go-lio-roy-messinger/>.
- [66] ADMIN, Confluence Wiki; O'NEAL, Terry. GPIO User Space App. In: *Xilinx Wiki* [online]. 14. 01. 2020 [cit. 2023-02-28]. Dostupné z: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842142/GPIO+User+Space+App>.
- [67] INC., Xilinx. Zynq-7000 SoC Technical Reference Manual (UG585). In: *Xilinx Documentation Portal* [online]. 02. 04. 2021 [cit. 2023-02-28]. Dostupné z: <https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM>.
- [68] THE MATHWORKS, Inc. Understanding PID Control, Part 2: Anti-windup for PID control. In: *The MathWorks, Inc. Videos and Webinars* [online]. [B.r.] [cit. 2023-03-04]. Dostupné z: <https://www.mathworks.com/videos/understanding-pid-control-part-2-expanding-beyond-a-simple-integral-1528310418260.html>.
- [69] XILINX, Inc. Stream Free Running Kernel (HLS C/C++). In: *GitHub* [online]. 2020 [cit. 2023-04-19]. Dostupné z: https://xilinx.github.io/Vitis_Accel_Examples/2020.2/html/streaming_free_running_k2k.html.

Příloha A: Seznam symbolů a zkratek

A.1 Seznam zkratek

AC	Alternating current
ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
API	Application Programming Interface
APU	Application Processor Unit
ASHW	Application Specific Hardware
ASIC	Application Specific Integrated Circuit
ASICs	Application Specific Integrated Circuits
ASM	Asynchronous Motor
AWS	Amazon Web Services
AXI	Advanced eXtensible Interface
BB	Base Board
BRAM	Block RAM
BSP	Board Support Package
BTN	Button
BUVH	bez ukládání a výpisu hodnot
CAN	Controller Area Network
CC	Carrier Card
CLK	Clock Signal
CMOS	Complementary metal-oxide-semiconductor
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DC	Direct Current
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DSP	Digital Signal Processing
DT	Device Tree
dtbo	Device Tree Blob Object
dtc	Device Tree Compiler
DTO	Device Tree Overlay
dtsi	Device Tree Source Include
DX	Developer Experience
EMC	Electromagnetic compatibility
FIFO	First In First Out
FOC	Field Oriented Control
FPD	Full Power Domain
FPGA	Field Programmable Gate Array
FPGAs	Field Programmable Gate Arrays
GIC	General Interrupt Controller
GND	Ground

GPIO	General-Purpose Input/Output
GPUs	Graphics Processing Units
GPUs	Graphics Processing Unit
GUI	Graphical User Interface
HATs	Hardware Attached On Top
HDL	Hardware Description Language
HIL	Hardware In the Loop
HLS	High Level Synthesis
HW	Hardware
I/O	Input/Output
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IOP	I/O peripherals
IP	Intellectual Property
IRQ	Interrupt Request
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LPD	Low Power Domain
LTS	Long Term Support
LUT	Look-Up Table
LUTs	Look-Up Tables
LVCmos	Low Voltage CMOS
LVDS	Low Voltage Differential Signaling
LVTTl	Low Voltage Transistor-Transistor Logic
MAP	Mapování v procesu konverze HDL na konfigurační bitstream
ML	Machine Learning
Mosi	Master Out
MPSoC	Multiprocessor System on a Chip
MUX	Multiplexer
PAR	Place-And-Route
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PI	Proporcionální Integrační
PL	Programmable Logic
PLA	Programmable Logic Arrays
PLD	Programmable Logic Devices
PMOD	PMOD Interface
PS	Processing System
PWM	Pulse Width Modulation

QSPI	Quad SPI
RAM	Random Access Memory
RK4	Runge–Kutta 4. řádu
ROM	Read Only Memory
RT	Real Time
RTL	Register Transfer Level
SBC	Single Board Computer
scp	secure copy
SD	Solid Drive
SDK	Software Development Kit
SFD	Single File Download
SFP	Small Form-factor Pluggable
SFTP	Secure File Transfer Protocol
SH	soubor hodnot
SoC	System on a chip
SOM	System on Module
SOMs	System on Modules
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
ssh	Secure Shell
SSTL	Stub Series Terminated Logic
SVM	Space Vector Modulation
SW	Switch
SW	Software
sw1	stav virtuálního spínače č. 1
sw2	stav virtuálního spínače č. 2
sw3	stav virtuálního spínače č. 3
TPUs	Tensor Processing Units
U+	Kladná polarita napětí U
UART	Universal Asynchronous Receiver-Transmitter
UIO	Userspace I/O
URAM	Unified Random Access Memory
USB	Universal Serial Bus
UVH	s ukládáním a výpisem hodnot
VE	Virtual Environment
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WIC	Windows Imaging Component
XDC	Xilinx Synopsys Design Constraints
xo	Xilinx Object
XPR	Vivado Project File
XRT	Xilinx Runtime
XSA	Xilinx Support Archive

A.2 Seznam symbolů

f_n	(Hz)	jmenovitá napájecí frekvence stroje
I_n	(A)	jmenovitý fázový proud stroje (efektivní hodnota)
I_1	(A)	velikost proudu statoru $i_1(t)$ procházející první fází do řízeného motoru, reprezentace v programu
$\underline{i}_1^{\alpha\beta}$	(A)	prostorový vektor statorového proudu v souřadnicovém systému $\alpha\beta$
$i_{1\alpha}$	(A)	složka vektoru statorového proudu v souřadnicovém systému spojeném se statorem stroje
$i_{1a}(t)$	(A)	proud statoru procházející první fází do řízeného motoru
$i_{1\beta}$	(A)	složka vektoru statorového proudu v souřadnicovém systému spojeném se statorem stroje
$i_{1b}(t)$	(A)	proud statoru procházející druhou fází do řízeného motoru
$i_{1c}(t)$	(A)	proud statoru procházející třetí fází do řízeného motoru
I_2	(A)	velikost proudu statoru $i_2(t)$ procházející druhou fází do řízeného motoru, reprezentace v programu
$\underline{i}_2^{\alpha\beta}$	(A)	prostorový vektor rotorového proudu v souřadnicovém systému $\alpha\beta$
I_3	(A)	velikost proudu statoru $i_3(t)$ procházející třetí fází do řízeného motoru, reprezentace v programu
\underline{i}_1^k	(A)	prostorový vektor statorového proudu v obecném souřadnicovém systému k
K	(-)	konstanta Clarkovi transformace
σ	(-)	rozptyl
L_1	(H)	statorová indukčnost
$L_{2\sigma}$	(H)	rotorová rozptylová indukčnost
L_1	(H)	rotorová indukčnost
L_m	(H)	magnetizační indukčnost
J	(kg·m ²)	moment setrvačnosti
M	(Nm)	vnitřní elektromechanický moment
M_z	(Nm)	zátěžný moment
n_n	(min ⁻¹)	jmenovité otáčky stroje
ω	(s ⁻¹)	elektrická úhlová rychlosť rotoru
Ω	(s ⁻¹)	mechanická úhlová rychlosť
ω_k	(s ⁻¹)	elektrická otáčivá rychlosť souřadnicového systému k
P_n	(W)	jmenovitý výkon stroje
p_p	(-)	počet polpárů stroje
$\underline{\psi}_2^k$	(Wb)	prostorový vektor rotorového toku v obecném souřadnicovém systému k

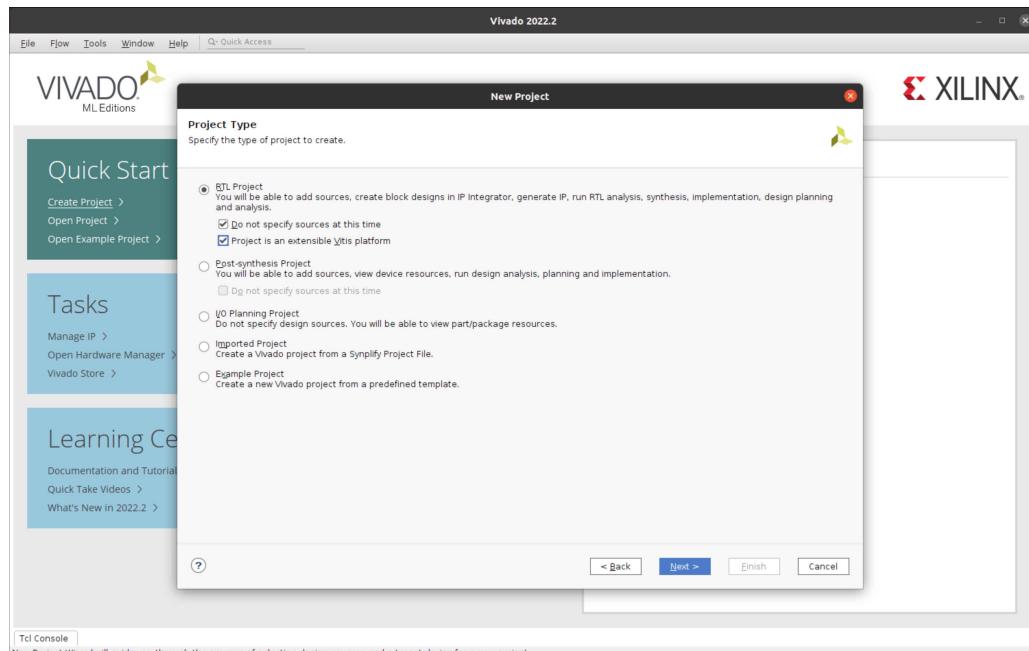
$\underline{\psi}_1^k$	(Wb)	prostorový vektor statorového toku v obecném souřadnicovém systému k
$\underline{\psi}_2^{\alpha\beta}$	(Wb)	prostorový vektor rotorového toku v souřadnicovém systému $\alpha\beta$
$\psi_{2\alpha}$	(Wb)	složka vektoru rotorového magnetického toku v souřadnicovém systému spojeném se statorem stroje
$\psi_{2\beta}$	(Wb)	složka vektoru rotorového magnetického toku v souřadnicovém systému spojeném se statorem stroje
ψ_2	(Wb)	velikost vektoru magnetického toku rotoru
$L_{1\sigma}$	(H)	statorová rozptylová indukčnost
R_1	(Ω)	rezistivita statorového vinutí
R_2	(Ω)	rezistivita rotorového vinutí
$sw1$	(1/0)	stav spínače č. 1 v trojfázovém invertoru
$sw2$	(1/0)	stav spínače č. 2 v trojfázovém invertoru
$sw3$	(1/0)	stav spínače č. 3 v trojfázovém invertoru
h	(-)	krok metody
t	(s)	čas
Θ	(rad)	transformační úhel pro Parkovu transformaci
U_n	(V)	jmenovité sdružené napájecí napětí stroje
$\cos(\varphi_n)$	(-)	jmenovitý účinník stroje
\underline{i}_1^k	(A)	prostorový vektor proudu statoru v obecném souřadnicovém systému k
\underline{i}_2^k	(A)	prostorový vektor proudu rotoru v obecném souřadnicovém systému k
\underline{u}_1^k	(V)	prostorový vektor napětí statoru v obecném souřadnicovém systému k
u_{1a}	(V)	velikost napětí statoru fáze A
$\underline{u}_1^{\alpha\beta}$	(V)	prostorový vektor napětí statoru v obecném souřadnicovém systému $\alpha\beta$
$u_{1\alpha}$	(V)	složka vektoru napětí statoru v souřadnicovém systému spojeném se statorem stroje
u_{1b}	(V)	velikost napětí statoru fáze B
$u_{1\beta}$	(V)	složka vektoru napětí statoru v souřadnicovém systému spojeném se statorem stroje
u_{1c}	(V)	velikost napětí statoru fáze C
\underline{u}_2^k	(V)	prostorový vektor napětí rotoru v obecném souřadnicovém systému k
$\underline{u}_2^{\alpha\beta}$	(V)	prostorový vektor napětí rotoru v souřadnicovém systému $\alpha\beta$

U_{dc} (V) napájecí napětí ve stejnosměrném meziobvodu inverteru

Příloha B: Tvorba HW designu pro Digilent Zybo Zynq-7000

V této části bude představen postup tvorby HW architektury pro vývojovou desku Digilent Zybo Zynq-7000. Postup tvorby platformy byl částečně převzat z [60]. V případě, že je požadováno vytvoření dalších speciálních bloků v PL je třeba do blokového designu vložit odpovídající IP bloky, které zajistí potřebnou funkcionalitu.

Nejprve je nutné vytvořit nový Vivado projekt a pojmenovat ho dle požadavků. Při výběru typu projektu je nutné zvolit možnost *RTL Project* a aktivovat možnost *Project is an extensible Vitis platform*. Tento úkon je naznačen na obr. B - 1.

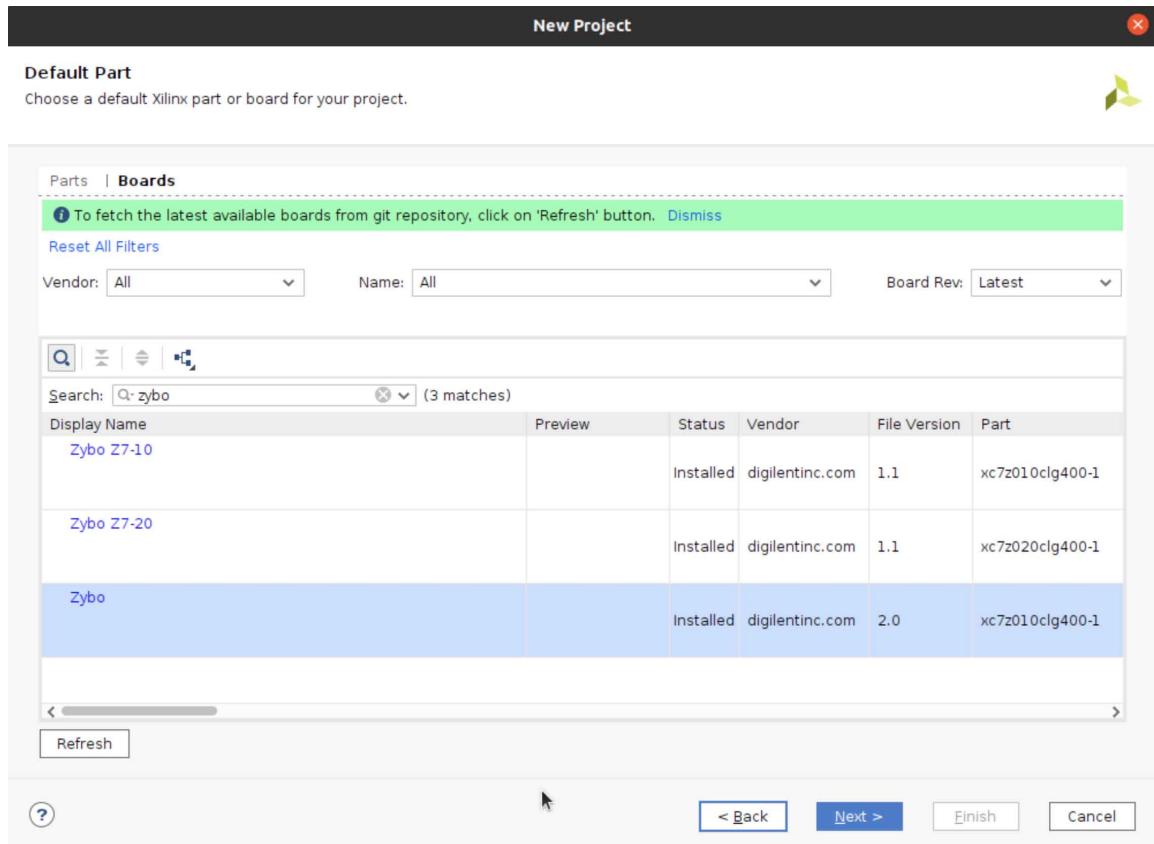


Obr. B - 1 Xilinx Vivado – volba typu projektu pro Digilent Zybo.

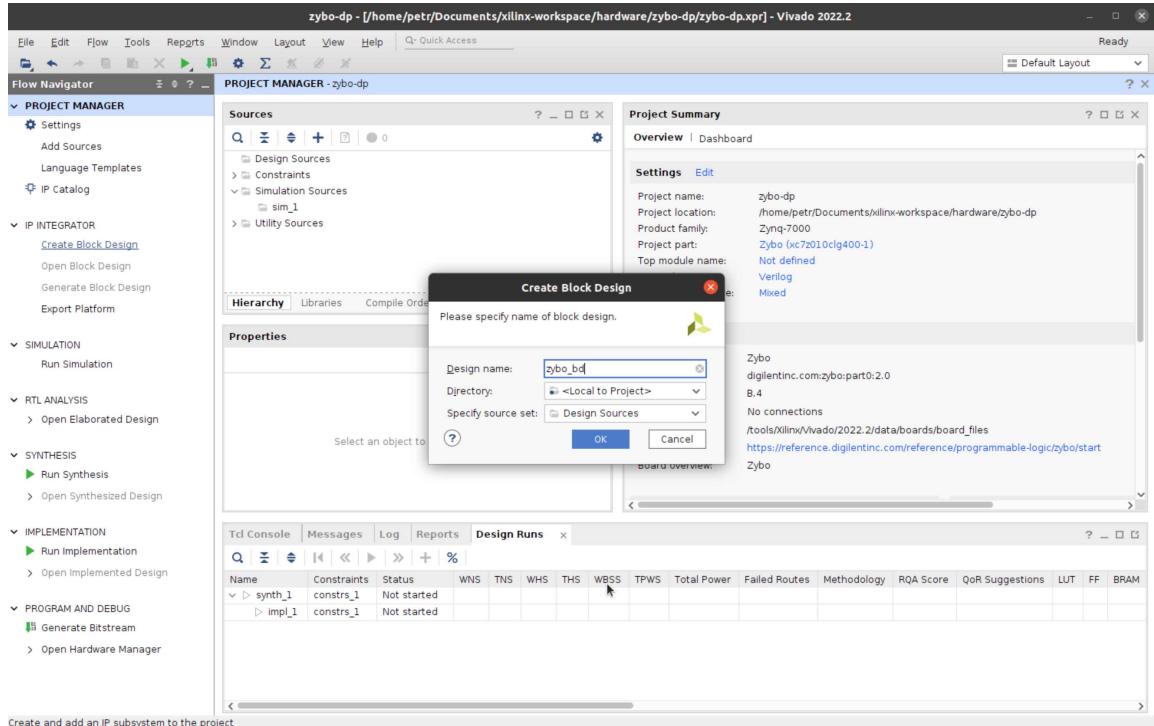
Následně v nabídce *Xilinx part* vybrat možnost *Board* a do vyhledávání zadat název využívané desky. Díky instalovaným *board files*, představených v části *Vivado Board Files*, je možné nalézt požadovanou desku Digilent Zybo verze 2.0 a pokračovat v tvorbě designu. Výběr základního HW je zobrazen na obr. B - 2.

Po úspěšné inicializaci projektu je pro další pokračování nutné v menu *Flow Navigator/IP Integrator* zvolit možnost *Create Block Design* a vytvořit nový blokový design. Tvorba blokového designu je naznačena na obr. B - 3.

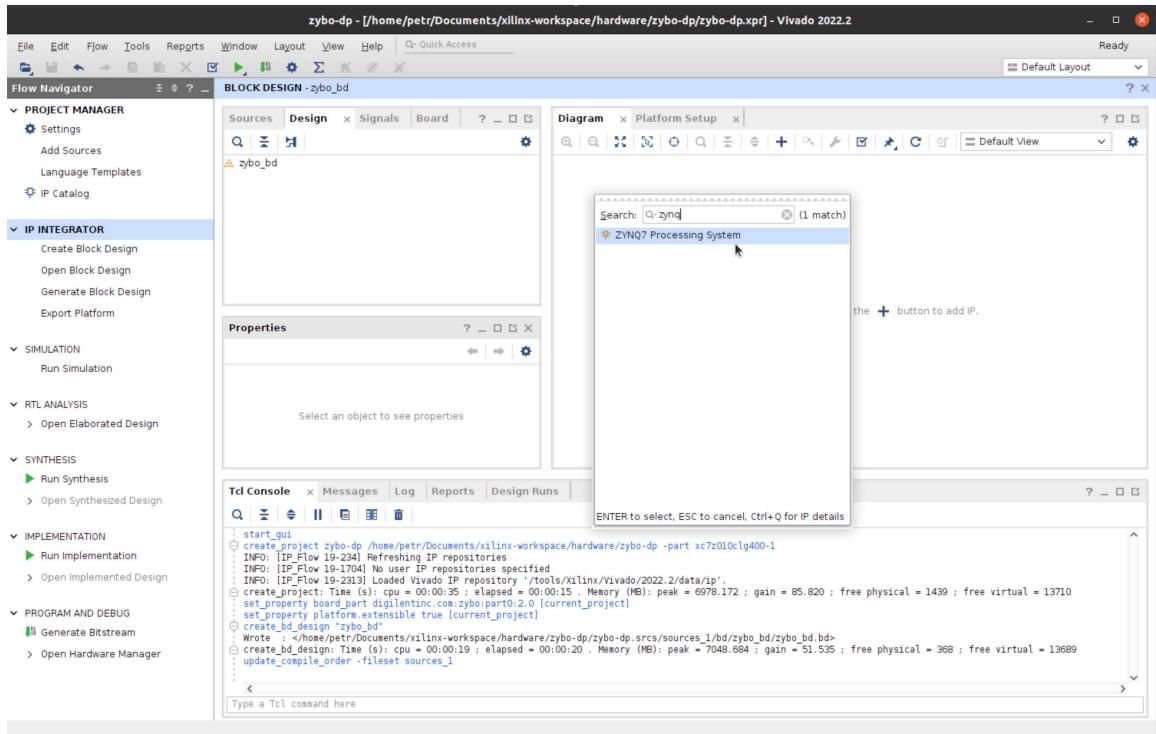
Nyní je již možné přistoupit k tvorbě vlastní architektury. Prvním krokem tvorby designu je vložit blok *ZYNQ7 Processing System* a zvolit nově zobrazenou možnost *Run Block Automation*. V těchto pomocných automatizacích je většinou výhodné ponechávat nastavené výchozí hodnoty, které jsou pro většinu tvořeného HW designu dostačující. Menu s výběrem IP bloku ZynQ PS je zobrazeno na obr. B - 4.



Obr. B - 2 Xilinx Vivado – výběr základního HW, pro který bude vytvářena architektura pro Digilent Zybo.



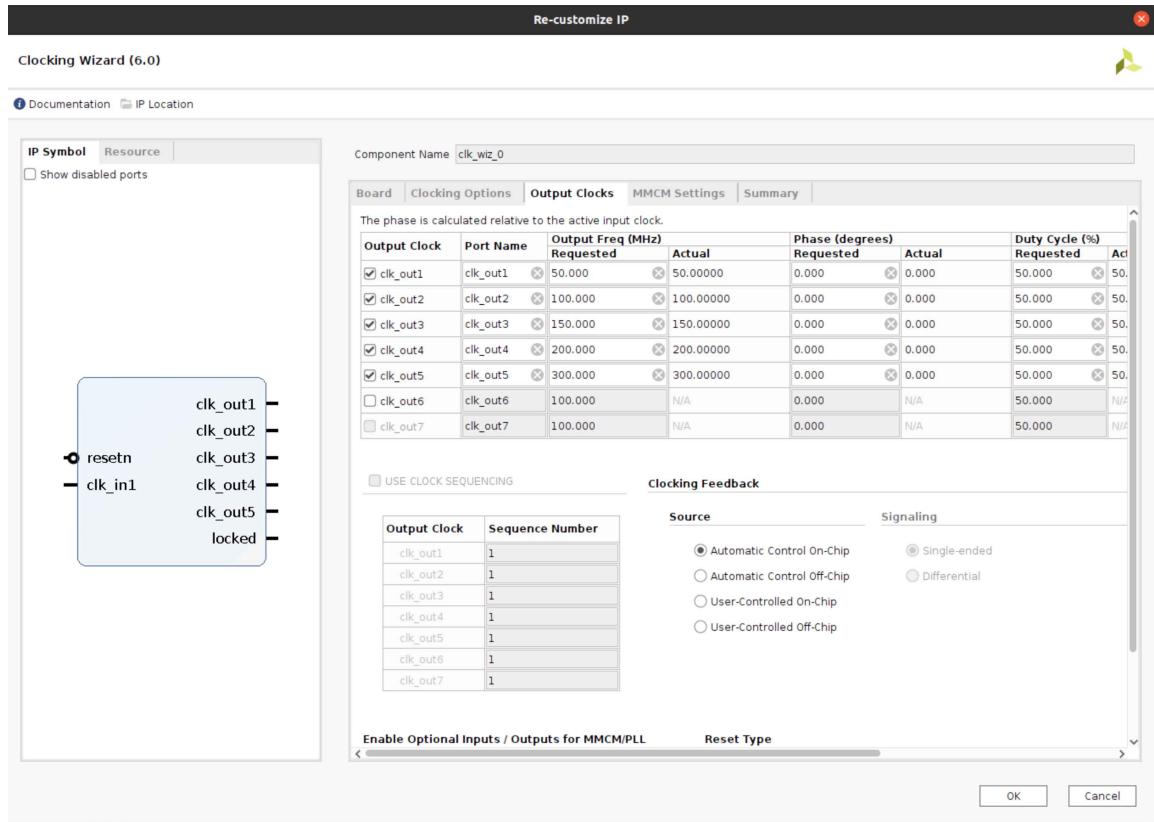
Obr. B - 3 Xilinx Vivado – vytváření Block Design pro Digilent Zybo.



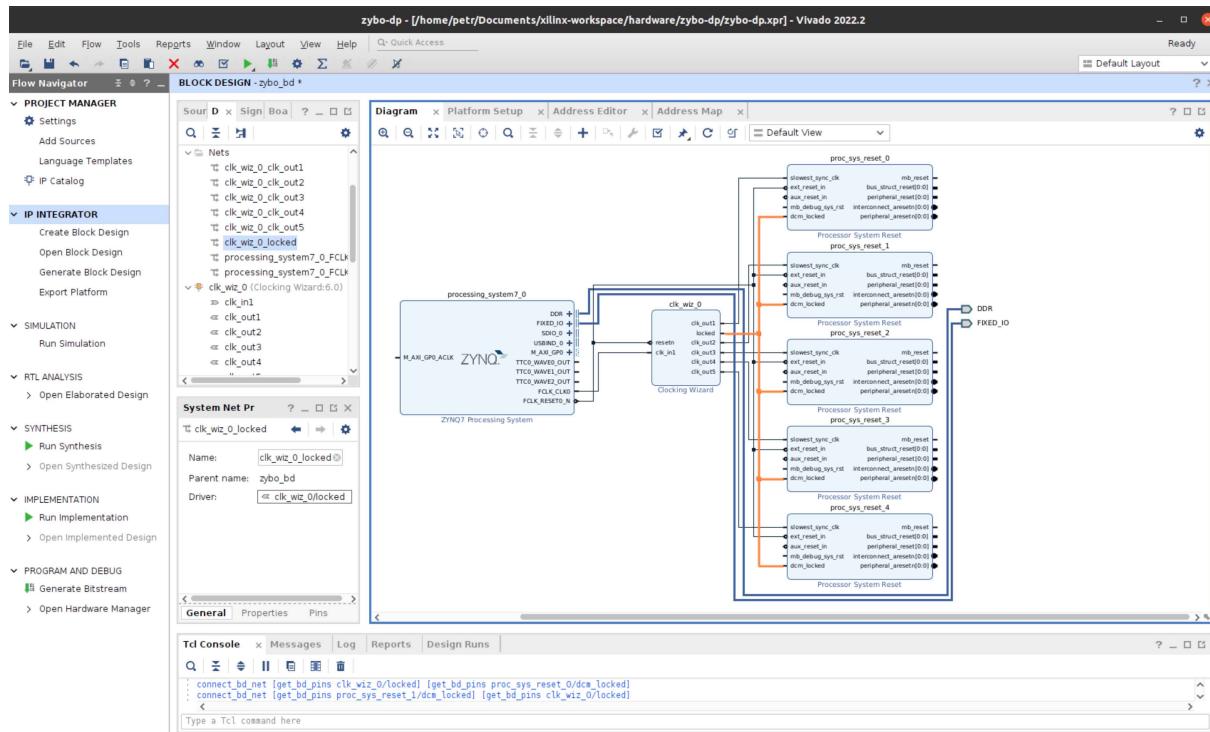
Obr. B - 4 Xilinx Vivado – vložení bloku ZYNQ7 Processing System pro Digilent Zybo.

Poté je pro funkční akcelerované aplikace nutné vložit do designu blok *Clocking Wizard*, ve kterém nastavit v záložce *Output Clocks*, aby byl signál aktivní v 0 a aktivovat pět výstupních signálů *Clock*. Těmto signálům je po aktivaci možné nastavit taktovací frekvenci na 50, 100, 150, 200 a 300 MHz. Poté je nutné na výstup *FCLK_CLK0* bloku *ZYNQ7 Processing System* připojit vstup *clk_in1* a k výstupu *FCLK_RESET0_N* vstup *resetn*.

Po nastavení bloku *Clocking Wizard* je zapotřebí do designu vložit pět bloků *Processor System Reset*. Následuje propojení odpovídajících výstupů bloků *Clocking Wizard* s názvem *clk_outX*, kde *X* značí pořadí výstupního signálu, s odpovídajícími bloky *Processor System Reset* a jejich vstupy *slowest_sync_clk*. Ke všem vstupům *dcm_locked* bloků *Processor System Reset* je nutné připojit výstup *locked* z *Clocking Wizard* bloku. A konečně ke všem vstupům *ext_reset_in* připojit výstup *FCLK_RESET0_N* ZynQ bloku. Představené propojení jednotlivých bloků je možné pozorovat na obr. B - 6.



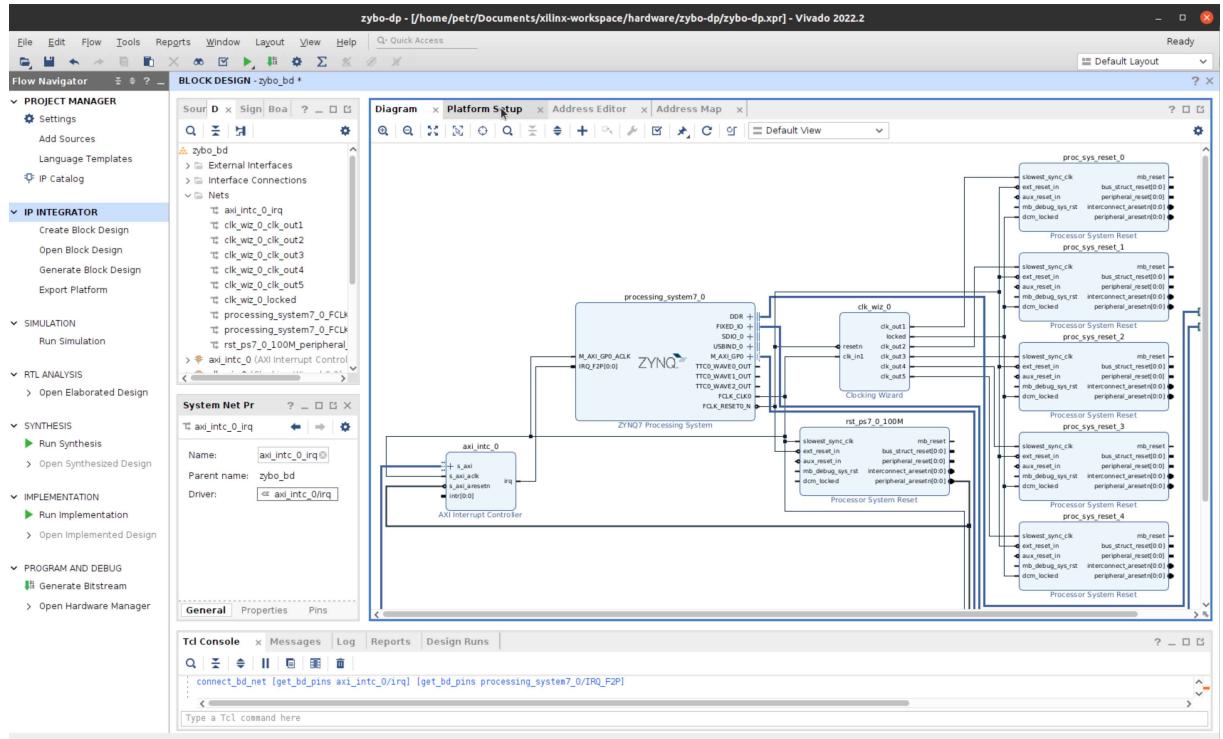
Obr. B - 5 Xilinx Vivado – nastavení výstupních taktovacích signálů pro Digilent Zybo.



Obr. B - 6 Xilinx Vivado – propojení bloků taktování pro Digilent Zybo.

Nyní je možné otevřít nastavení *ZYNQ7 Processing System* a v záložce *Interrupts* povolit nastavení *Fabric Interrupts/PL-PS Interrupt Ports/IRQ_F2P* přerušení. Následně do designu je nutné vložit blok řídící přerušení *AXI Interrupt Controller*, otevřít jeho nastavení a v sekci *Processor Interrupt Type and Connection* změnit nastavení *Interrupt Output Connection* z *Bus* na *Single*. Poté je opět možné spustit automatické propojení jednotlivých bloků s výchozím nastavením.

Aby bylo přerušení funkční, je třeba propojit výstup bloku *AXI Interrupt Controller irq* se vstupem bloku *ZYNQ7 Processing System IRQ_F2P*. Minimální funkční blokový design je zobrazen na obr. B - 7.



Obr. B - 7 Xilinx Vivado – minimální funkční blokový design pro akcelerovanou aplikaci pro Digilent Zybo.

Nyní je možné přejít ze záložky *Diagram* do záložky *Platform Setup*. Následuje nastavení parametrů bloku *ZYNQ7 Processing System* v záložce *AXI Port* dle tabulky č. B - 1.

Tab. B - 1 Ukázka nastavených AXI portů v Xilinx Vivado platformě pro Digilent Zybo.

Name	Enabled	Memport	SP Tag
M_AXI_GP1	X	M_AXI_GP	-
S_AXI_ACP	O	-	-
S_AXI_HP0	X	S_AXI_HP	HP0
S_AXI_HP1	X	S_AXI_HP	HP1
S_AXI_HP2	X	S_AXI_HP	HP2
S_AXI_HP3	X	S_AXI_HP	HP3

Aby bylo možné zapisovat do globální paměti přes MAXI Adapter je nutné povolit funkci vybraných portů v bloku *AXI Interconnect*. V této práci byly povoleny porty *M01_AXI* až *M32_AXI*.

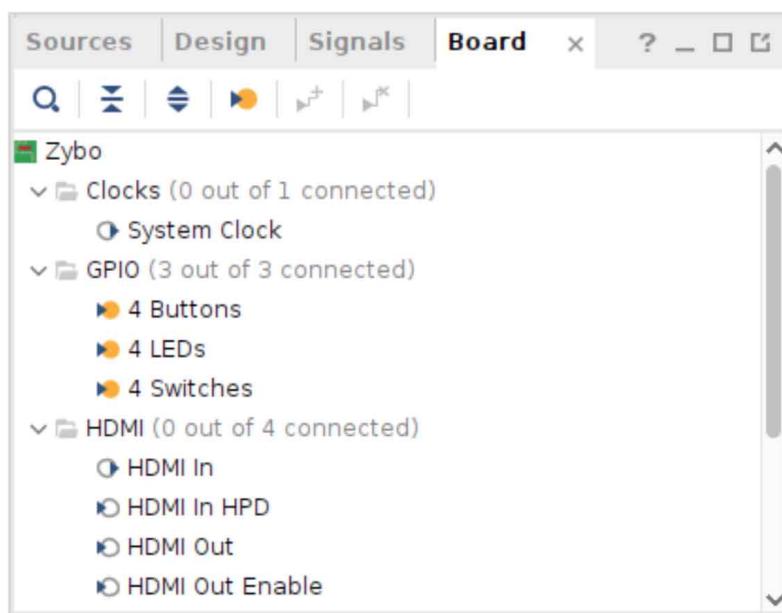
Následně v záložce *Clock* je nutné povolit *clk_outx*, kde $x \in \{1, 5\}$, nastavit jejich odpovídající ID a jako výchozí použít taktovací signál 100 MHz.

Dále je v záložce *Interrupt* nutné aktivovat výstup *intr* bloku *AXI Interrupt Controller*.

Aby bylo možné případně provádět HW-emulaci, je nutné v kartě *Diagram* zvolit blok *ZYNQ7 Processing System* a v záložce *Block Properties* v nabídce *SELECTED_SIM_MODEL* zvolit možnost tlm. [60]

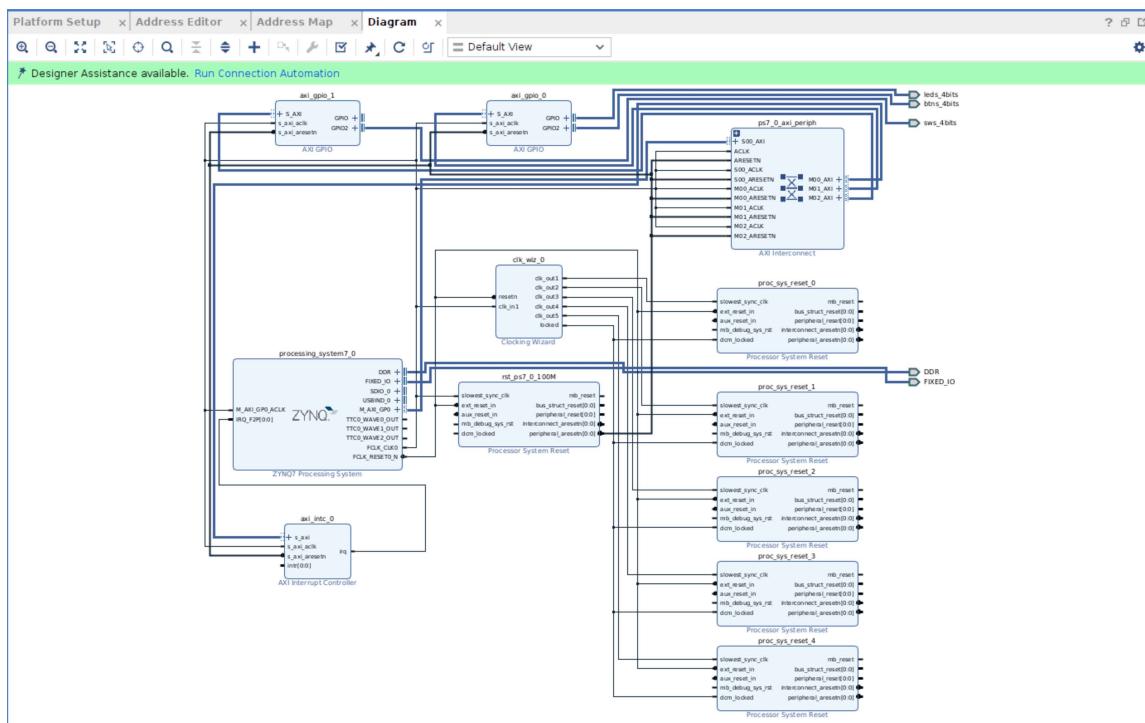
V této práci nebylo dosaženo funkční SW ani HW simulace pro desku Digilent Zybo.

Pro zajištění funkčních GPIO (General Purpose Input/Output) pro ovládání LED signalizace, tlačítek a přepínačů, připojených na PL, je nutné do blokového designu z karty *Board/Zybo/GPIO* vložit potřebné IP. Po kliknutí pomocí pravého tlačítka myši na vybraný prvek je nutné zvolit z nabídky *Connect board component* požadovaný GPIO interface. Blok *AXI GPIO* je automaticky vložen do blokového designu a jeho přítomnost je signalizována na kartě *Board/Zybo/GPIO* zvýrazněním vložených bloků (ukázka na obr. B - 8). Dle požadavku uživatele je možné vybrat, zda bude využíváno *GPIO* nebo *GPIO2* rozhraní. Dle vybraného rozhraní budou poté adresovány jednotlivé výstupy a vstupy v *PetaLinux*. Ukázka vytvořeného designu s IP pro GPIO je na obr. B - 9.

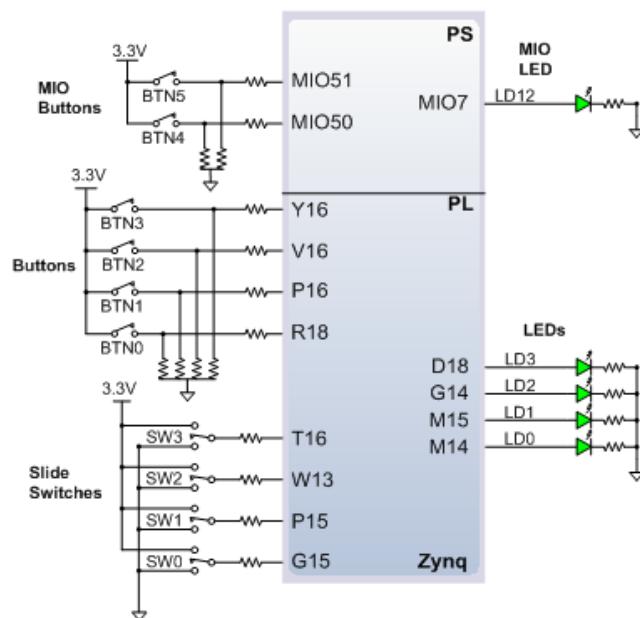


Obr. B - 8 Xilinx Vivado – signalizace vložených AXI GPIO bloků pro LED, BTN, SW na kartě Board/Zybo/GPIO pro Digilent Zybo.

Rozdělení vstupů a výstupů do jednotlivých částí SoC – PS a PL je vyzobrazeno na obr. B - 10.



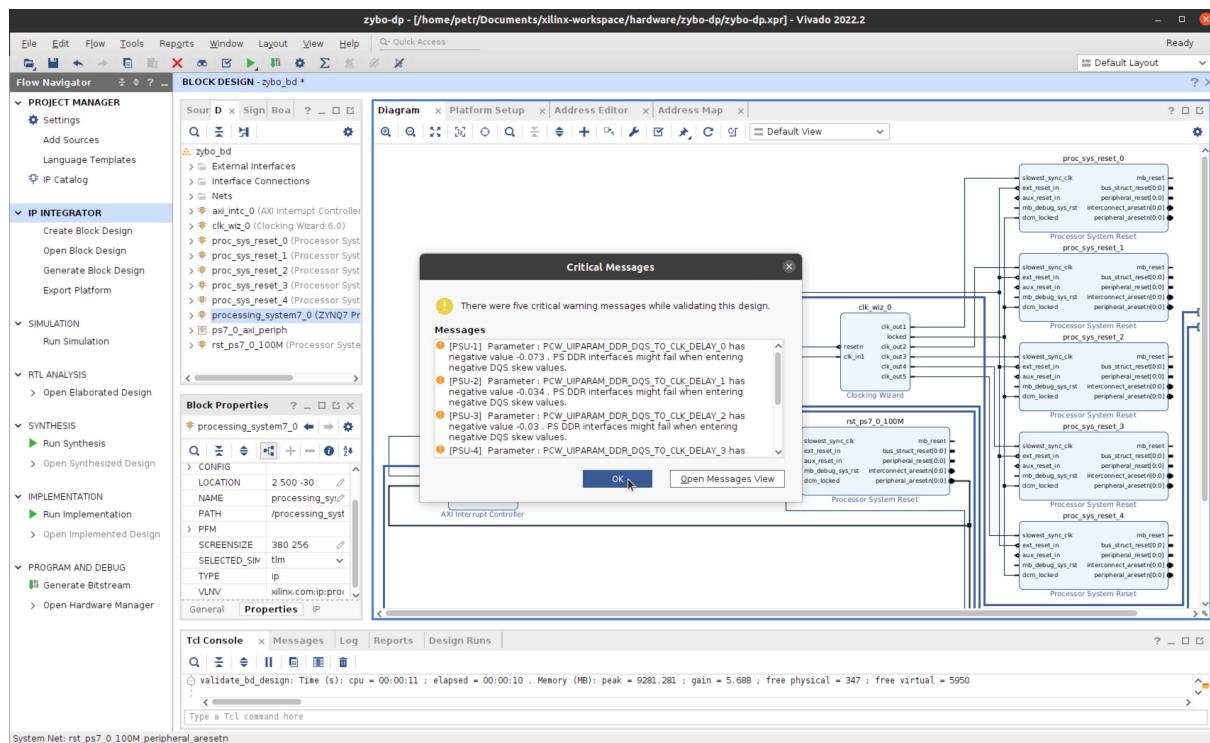
Obr. B - 9 Xilinx Vivado – block design s využitím GPIO pro LED, BTN, SW propojených s PL pro Digilent Zybo.



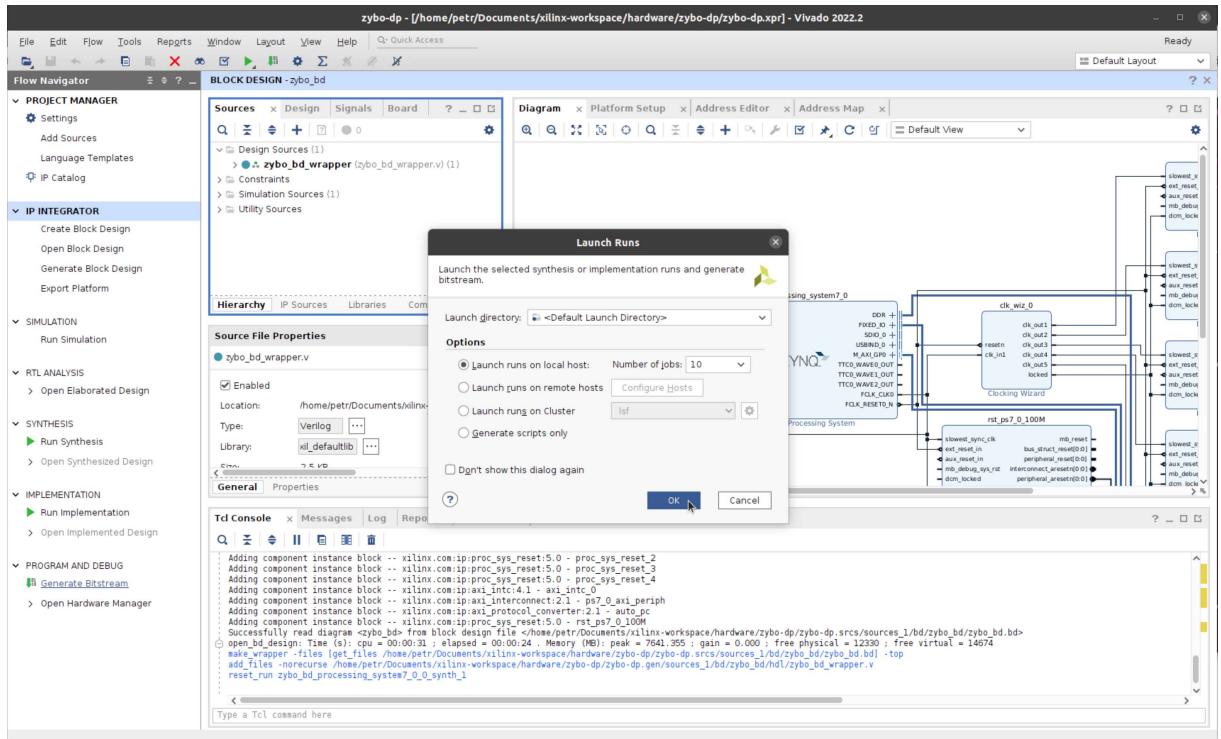
Obr. B - 10 Uspořádání připojení tlačítek, přepínačů a LED k PS a PL pro vývojovou desku Digilent Zybo. [24]

Před dalším pokračováním je možné design validovat pomocí příslušného tlačítka validace na horizontální ovládací liště. Pokud je již HW design vytvořen a nakonfigurován, je možné v kartě *Sources/Design Sources* vybrat vytvořený design a pomocí nabídky pravého tlačítka myši vybrat možnost *Create HDL Wrapper*. V tomto kroku je opět prováděna validace designu. Pokud se v designu vyskytuje pouze kritická upozornění, která jsou zobrazena na obr. B - 11, je stále možné pokračovat v tvorbě.

Po vytvoření *HDL Wrapper* je možné v menu *Flow Navigator/Program and Debug* zvolit krok *Generate Bitstream*. Pokud do tohoto kroku nebyla provedena syntéza ani implementace designu, objeví se hlášení, že je třeba tyto kroky provést. V případě pokračování v požadavku na generování bitstreamu budou automaticky kroky syntézy a implementace provedeny. V navazující nabídce je možné vybrat, zda procesy budou probíhat lokálně či na vzdáleném serveru, nebo clusteru. Také je možné zvolit počet výpočetních jader procesoru, který se bude podílet na prováděných úkolech. V případě využití osobního počítače pro generaci bitstreamu (i předcházející syntézy a implementace) autor práce doporučuje používat méně než polovinu dostupných jader. Tato volba vychází z experimentálního zjištění, že v případě využití vyššího počtu jader může dojít k neočekávané chybě a proces provádění úkolů bude bez udání jakékoli informace ukončen. Tudíž proces syntézy, implementace a generace bitstreamu bude nutné spustit znova. Ukázka nastavení procesu je zobrazena na obr. B - 12.



Obr. B - 11 Xilinx Vivado – kritická upozornění, zobrazená po validaci designu, která je možné ignorovat.



Obr. B - 12 Xilinx Vivado – nastavení provádění úkonů syntézy, implementace a generování bitstreamu, volba použitých výpočetních jader a určení, kde se mají procesy vykonávat pro Digilent Zybo.

Indikátor provádění jednotlivých procesů je umístěn v pravém horním rohu. Záznam prováděných procesů je umístěn v kartě *Log*.

Po úspěšném provedení jednotlivých kroků je zobrazena nabídka, která umožňuje nahlédnout na vytvořený design. Tuto nabídku je možné uzavřít aniž by byla vykonávána jakákoli z nabízených možností.

Po ukončení procesů je pro použití vytvořeného designu k tvorbě *PetaLinux* systému a aplikace v Xilinx Vitis IDE nutné exportovat vytvořenou platformu. Export je proveden pomocí volby *File/Export/Export platform*. Ve výběru platformy je výhodné zvolit možnost *Hardware and hardware emulation*, která umožňuje použít design pro skutečný HW i jeho emulaci. V nabídce *Platform State* je nutné vybrat možnost *Pre-Synthesis*, která umožňuje další zpracování platformy v Xilinx Vitis IDE pomocí v++.

Důležitou volbou je vybrání možnosti *Include bistream*, který byl produktem tvorby designu v této části.

Po nastavení dodatečných informací platformy je možné spustit export do požadované lokace.

Příloha C: Upravený postup nahrávání aplikace pro Digilent Zybo

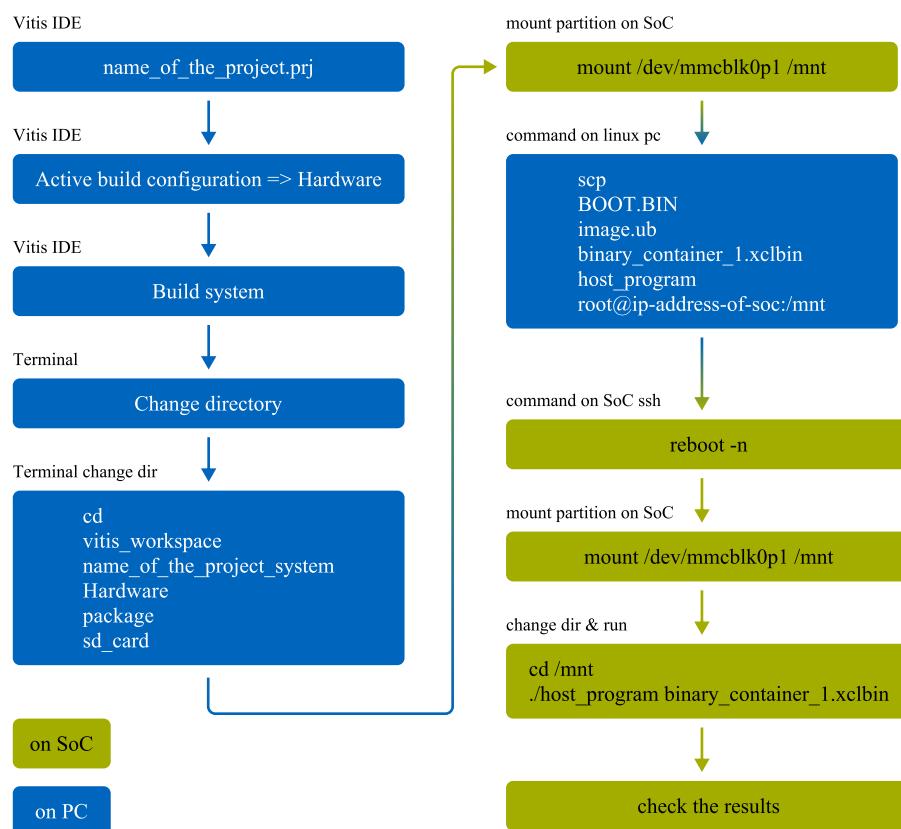
Při prvotním zkoumání vývoje aplikací pro Digilent Zybo byl objeven postup, kterým kontrolovat jejich funkčnost. Tento postup je naznačen na obr. C - 1.

Po nastavení build procesu ve Vitis IDE tak, aby vytvářel spustitelnou aplikaci na HW, je možné po spuštění zařízení připojit partition *mmcblk0p1*. Následně je možné do prostoru zařízení přesunout soubory:

- BOOT.BIN,
- image.ub,
- <binary-container-name>.xclbin,
- <host-program-name>.

Tyto soubory jsou přesunuty z vývojářského zařízení, které je v části *Popis pracoviště* v obr. 16 - 1 označeno jako **development machine**. Přesun souborů je možné provést pomocí scp.

Po úspěšném přesunu souborů je nutné zařízení Digilent Zybo restartovat a opět připojit partition. Následně je již možné spustit host aplikaci na PS a nakonfigurovat PL pomocí **xclbin** souboru.



Obr. C - 1 Diagram popisující upravený postup pro nahrávání a spouštění akcelerované aplikace pro Digilent Zybo.

Příloha D: Python skript pro vykreslení výsledných průběhů simulace

V kódu D - 1 je uveden Python skript, který slouží k vizualizaci dat, získaných ze simulace v části *CPU/FPGA Model*.

```
1 # importing dependencies
2 # GRAPH USED IN A~CONJUNCTION WITH testCalculationLoop file program
3 import pandas as pd
4 from matplotlib import pyplot as plt
5
6 # setting global font for plots
7 plt.rcParams["font.family"] = "Times New Roman"
8
9
10 # defining custom CTU FEE colors
11 ctuBlue="#0065BD"
12 ctuLightBlue = "#6AADE4"
13 ctuRed = "#C60C30"
14 ctuGreen = "#A2AD00"
15 ctuGreenyBlue = "#00B2A9"
16 ctuOrange = "#E05206"
17 ctuYellow = "#F0AB00"
18
19
20 # setting figure size, the value is in inches, but help from one tutorial
# gives info, that because of dpi it translates to pixels like inches *
# 100 = pixels
21 plt.rcParams["figure.figsize"] = [15, 6]
22
23 # some kind of autolayout
24 plt.rcParams["figure.autolayout"] = True
25
26 # defining which columns are in imported CSV
27 columns = ["globalSimulationTime", "psi2amplitude", "
    motorMechanicalAngularVelocity", "idRegulatorMeasuredValue", "
    idRegulatorWantedValue", "velocityRegulatorWantedValue", "
    velocityRegulatorSaturationOutput", "velocityRegulatorClampingStatus", "
    fluxRegulatorISum"]
28 df = pd.read_csv("./globalSimulationData.csv", names=columns, header=None,
    skiprows=0, nrows=10000000)
29
30 # print out part of the csv files as a text to terminal
31 print("Contents in csv file:", df)
32
33 figure_first = plt.figure("psi2amplitude")
34
35 plt.title("Závislost mechanické čotáivé rychlosti rotoru a velikosti
    magnetického toku rotoru na čase", fontsize=22, fontweight=700)
```

```

36
37 # ax = axis, but subplot is here mainly for deleting some axis on top and
38 # right
39 ax = plt.subplot(111)
40 ax.spines[['right', 'top']].set_visible(False)
41 # formating ticks, there can be used fontweight="bold"
42 plt.yticks(fontsize=15, fontweight=700)
43 plt.xticks(fontsize=15, weight=700)
44
45 ax.xaxis.get_offset_text().set_fontsize(18)
46 offset_text = ax.xaxis.get_offset_text()
47 offset_text.set_visible(False) # set offset text not visible
48 offset_text.set_position((0, 0)) # move horizontally to the center and
49 # down by
50
51 # adding grid
52 plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
53
54 plt.plot(df.globalSimulationTime, df.psi2amplitude, color=ctuBlue, label="psi2amplitude")
55 plt.plot(df.globalSimulationTime, df.motorMechanicalAngularVelocity, color=ctuRed, label="angularVelocity")
56 plt.xlabel("time (s)", fontsize=17, fontweight=400, loc = "right")
57 plt.ylabel("psi2amplitude (Wb)\nangularVelocity (s^(-1))", fontsize=14 ,
58 fontweight=400, loc = "top", rotation=0)
59 plt.legend( bbox_to_anchor=(0.5, -0.05), ncol=2, fontsize=14 )
60 plt.show()

```

Kód D - I Ukázka python skriptu pro vykreslení časové závislosti mechanické otáčivé rychlosti rotoru Ω a velikosti magnetického toku rotoru ψ_2 .