



# **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: Ing. Jan Bauer, Ph.D.

**Petr Zakopal  
Praha 2023**



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zakopal** Jméno: **Petr** Osobní číslo: **483802**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/ústav: **Katedra elektrických pohonů a trakce**  
Studijní program: **Elektrotechnika, energetika a management**  
Specializace: **Aplikovaná elektrotechnika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Oživení pracoviště s měničem DCM a PLC SIMATIC**

Název bakalářské práce anglicky:

**Workpalce with Rectifier DCM and PLC SIMATIC**

Pokyny pro vypracování:

- 1) Seznamte se s měničem řady DCM firmy SIEMENS
- 2) Oživte základní regulační smyčky měniče (otáčkovou, proudovou)
- 3) Prostudujte možnosti záznamu průběhů z měniče pomocí PLC nebo dotykového panelu
- 4) Pomocí PLC SIMATIC S1200 a dotykového panelu realizujte vzdálené ovládání a monitoring měniče
- 5) Na dotykovém panelu vytvořte obrazovku pro nastavování otáček nebo momentu motoru napájeného měničem

Seznam doporučené literatury:

- [1] Weidauer J., Messer R. Electrical Drives, Publics Erlangen, 2014
- [2] SCE Training Curriculum. Siemens AG, 2016
- [3] Durry B. The Control Techniques Drives and Controls Handbook 2nd ed., IeT, 2009
- [4] Pavelka J., Kobrle P. Elektrické pohony a jejich řízení. 3. prepracované vydání. Praha: České vysoké učení technické v Praze, 2016. ISBN 978-80-01-06007-0.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Jan Bauer, Ph.D., katedra elektrických pohonů a trakce FEL**

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

Datum zadání bakalářské práce: **24.01.2021**

Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Jan Bauer, Ph.D.  
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.  
podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání

Podpis studenta



## 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 Zákapal

## PODĚKOVÁNÍ

Nam quis commodo justo. Mauris diam metus, mattis sed rutrum in, volutpat sit amet sem. Nam bibendum commodo porttitor. Quisque eget lectus rutrum, molestie tortor id, iaculis nunc. Sed et maximus ipsum. Vivamus vel facilisis nisl. Curabitur eu nibh nec erat mollis finibus at in sapien. Mauris viverra sapien neque, nec lacinia odio laoreet eu. Quisque consectetur eros ac orci interdum scelerisque.

## **ABSTRAKT**

Nam quis commodo justo. Mauris diam metus, mattis sed rutrum in, volutpat sit amet sem. Nam bibendum commodo porttitor. Quisque eget lectus rutrum, molestie tortor id, iaculis nunc. Sed et maximus ipsum. Vivamus vel facilisis nisl. Curabitur eu nibh nec erat mollis finibus at in sapien. Mauris viverra sapien neque, nec lacinia odio laoreet eu. Quisque consectetur eros ac orci interdum scelerisque.

**Klíčová slova:** Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis aliquam finibus sagittis. Nunc venenatis, augue quis luctus dictum, elit justo pharetra leo, nec viverra purus dui at quam.

## **ABSTRACT**

Nam quis commodo justo. Mauris diam metus, mattis sed rutrum in, volutpat sit amet sem. Nam bibendum commodo porttitor. Quisque eget lectus rutrum, molestie tortor id, iaculis nunc. Sed et maximus ipsum. Vivamus vel facilisis nisl. Curabitur eu nibh nec erat mollis finibus at in sapien. Mauris viverra sapien neque, nec lacinia odio laoreet eu. Quisque consectetur eros ac orci interdum scelerisque.

**Keywords:** Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis aliquam finibus sagittis. Nunc venenatis, augue quis luctus dictum, elit justo pharetra leo, nec viverra purus dui at quam.

## **OBSAH**

## **SEZNAM OBRÁZKŮ**

## **SEZNAM TABULEK**





# 1 Úvod

V době, kdy byla od elektrických pohonů požadována spolehlivost, vysoká účinnost a nenáročné ovšem kvalitní řízení, byly k řízení využívány samotné digitální signálové procesory. Postupem času dochází ke zjištění, že výkon DSP není dostatečný a na některé aplikace, kde je vyžadováno provedení značného množství náročných výpočtů za co nejkratší čas, nejsou vhodné. Proto nastupuje éra logických programovatelných polí (FPGA), které jsou schopny tyto výpočty provést s velmi nízkými nároky na energii a za velmi krátký čas.

V mnoha odvětvích se již začíná využívat embedded systém s Application Specified Hardware, který je určen pouze na využití v předem dané aplikaci. Tento hardware slouží v dané aplikaci k jedinému účelu, který vykonává a na který je optimalizován. Tím se liší od procesoru, který vykonává mnoho instrukcí a využít ho pouze jako samostatnou výpočetní jednotku je z hlediska energetické i finanční náročnosti nevýhodné. Implementace hradlových polí přináší nejen v řízení elektrických pohonů zvýšení výpočetního výkonu, ale také snižování energetické náročnosti řízení.

Perspektiva logických programovatelných polí a hardwaerově urychlovaných aplikací je podpořena jejich využíváním i mimo obor elektrických pohonů a trakce. Z důvodu jejich veliké propustnosti, vysokých výpočetních výkonů a nízké energetické náročnosti jsou využívány v AI, machine learningu, zpracování obrazu, těžení kryptoměn a jiných nepohonářských aplikacích.

Nevýhodou problematiky FPGA je jejich složitější programovatelnost z hlediska tvoření aplikace. Aplikace je tvořena určitým postupem (workflow), který kladne vysoké nároky na vzdělání a zkušenosti vývojářů. Většina FPGA je programována pomocí jazyků Verilog či VHDL, které mohou pro softwarově orientované programátory představovat značnou překážku. Proto bylo vyvinuto tvoření aplikací pomocí vyšší úrovně syntézy (HLS), kdy je možné tvořit programy ve vyšších programovacích jazycích jako je například C, C++ či Python. HLS umožnilo rapidní rozšíření a využití Embedded FPGA Accelerated Applications v mnoha aplikacích a značně vylepšilo vývojářský požitek (developer experience, DX) při tvorbě aplikací.

Protože může být náročné vytvořit vlastní architekturu, složenou z CPU a spolupracujícího FPGA, je vhodné při prvotním vývoji aplikace využít dostupné vývojové desky obsahující již předpřipravené propojení jednotlivých komponent. Součástí těchto vývojových desek bývá také mnoho vstupů a výstupů (I/O) pro snadnější využití při lazení a tvoření aplikace. V této práci je využívána vývojová deska Zybo od firmy Digilent. Ovšem autor v textu představuje další možnosti, které mohou být pro konkrétní aplikace a využití vhodnější.

Tato práce se zajímá o aplikace a možné využití FPGA při řízení elektrických pohonů. Autor v ní představuje základní principy Hardware Accelerated Applications, z jakého důvodu je tento přístup perspektivní a proč je vhodné se orientovat tímto směrem.

## 2 System on a chip

System on a chip (SoC) je architektura čipu, využívající takovou konstrukci, kdy jsou integrovány různé části/bloky systému na jeden čip. Integrace prvků na jeden čip značně sniže nároky na rozměry nosičů, na kterých jsou tyto SoC umístěny. Místo diskrétních čipů, obstarávající jednotlivé funkce, je využito jednoho čipu s mnoha částmi vykonávající požadované funkce.

Protože integrování čipů do jedné polovodičové struktury představuje relativně veliké snížení nároků na kovové vodivé spoje a sniže časovou náročnost a zvyšuje rychlosť přenosu dat, je SoC upřednostňováno před metalicky spojenými diskrétními částmi vykonávajícími dané operace.

Označení SoC může představovat mnoho architektur. Obecné rozdělení, nalezitelné v literatuře a veřejných zdrojích, těchto architektur je následující:

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

Jedná se tudíž o rozdělení dle hlavní výpočetní jednotky, resp. procesoru v čipu. [[tomshardware-system-on-chip](#)]

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 využíváno přesvědčení, že pokud je architektura HW přímo specializovaná na jednu aplikaci, 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, výpočtů apod. Tyto ASIC mohou také vykonávat potřebné rychlé výpočty pro matematické modely elektrických strojů, které jsou využívány např. pro HIL.

Než je tento specifický obvod vytvořen, je nutné jej navrhnout, vyzkoušet a odladit. K tomu slouží logická programovatelná pole, ve kterých je možné požadovaný HW navrhnout a odladit před velkou produkcí ASIC. Pokud velká produkce není z ekonomických důvodů možná, jsou FPGA využívány přímo v produkci s jejichž pomocí 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. Nejvýznamnější využití v problematice elektrických pohonů je v embedded systémech a hardwaerově akcelerovaných aplikacích.

### 3 System on Modules

System on modules (SOMs) je architektura jejíž jednou z hlavních součástí je dříve zmiňovaný SoC. SOMs se oproti SoC již dodávají na PCB a kromě SoC mají na desce umístěné další komponenty, které jsou pro danou aplikaci vyžadovány. [[xilinx-what-is-a-som](#)]

SOMs se mohou dodávat jako vývojové desky [[xilinx-kria-kr260-robotics-starter-kit](#)], které obsahují krom SOM také podpůrnou desku s dalšími obecnými komponenty, jež jsou vhodné pro vývoj standardních aplikací. Pokud zákazník již pomocí vývojové desky odladil vytvářenou aplikaci, může zakoupit samostatný SOM na base board (BB) a podpůrnou desku (tzv. carrier card, CC) pro danou aplikaci navrhnu tak, aby obsahovala pouze komponenty, které daná aplikace využívá. Tudíž se snižuje cena konečného výrobku o komponenty, které byly z carrier card při návrhu odstraněny pro jejich nevyužití.

Výrobce platformy *Kria KR260 Robotics Starter Kit*, použité v této práci, dodává k produktu rozsáhlou dokumentaci [[kria-som-carrier-card-design-guide-2022](#)] [[kria-k26-som-ds](#)], podle které je možné individuální carrier card sestavit.

Příkladem individuálně vytvořené carrier card je open source projekt od firmy *Antmicro Ltd.*. Tato firma vydala open source design carrier cardu pro zařízení *Kria K26*, které je předchůdcem *KR260*, a bylo určeno pro akcelerování audiovizuálních aplikací. Využívá však totičný SOM ale rozdílný carrier card. Dokumentace a vytvořený návrh je dostupný z [[antmicro-open-source-kria-k26-carrier-card](#)].

#### 3.0.1 Embedded Systems

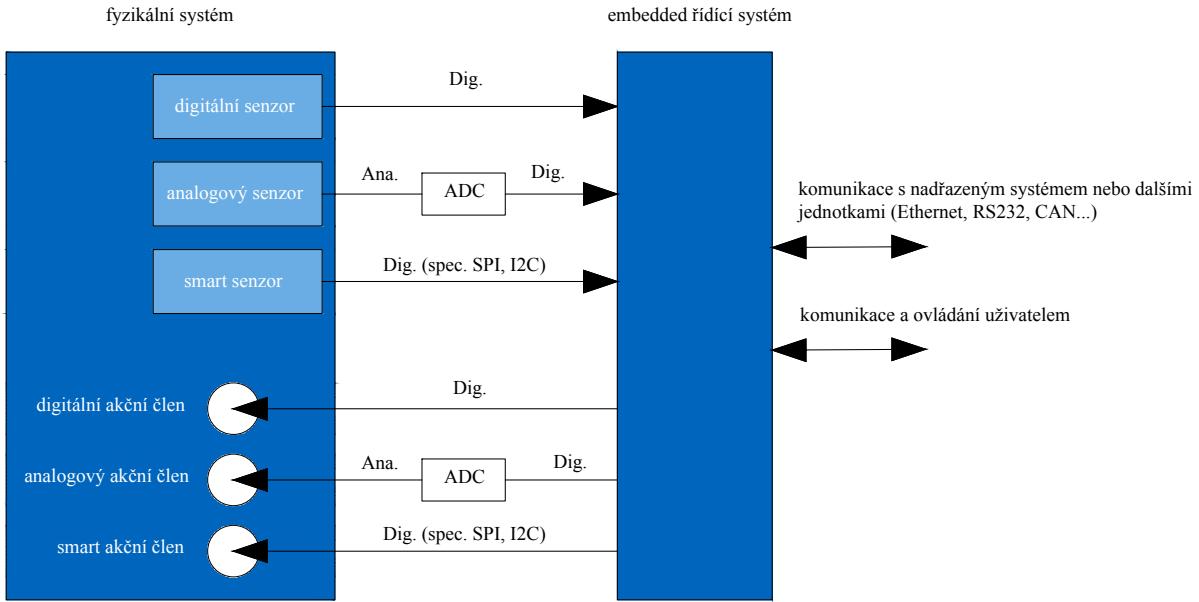
Embedded systéms je název pro skupinu zařízení, obecně systémů, které je možné charakterizovat jako specifické výpočetní zařízení, resp. počítače, které jsou určeny pro podporu funkce nebo řízení nějakého většího celku, produktu nebo fyzikálního systému. Oproti tomu osobní počítač je sice výpočetní zařízení, ale nelze mluvit o embedded systému, protože je určen pro mnoho univerzálních aplikací. [[Sass2010](#)]

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. [[Sass2010](#)] [[juan-fpgas](#)]

Do embedded systému vstupují signály, které jsou následně zpracovány a poté vybrané výsledky výpočtu jsou v podobě výstupní signálů výstupním produktem systému. Tyto produkty mohou pomoci akčním členů zasahovat do řízeného systému. Vstupní signály většinou přicházejí ze speciálních snímačů, kompatibilních s embedded systémem (senzor teploty, senzor tlaku, senzor zrychlení, gyroskop, senzory proudu, inkrementální čidla apod.). Naopak jeho výstupní signály jsou například specifická ovládací hodnota napětí, proudu nebo jiné veličiny. Také mohou být na výstupních pinech připojené LED signalizace, komunikační sběrnice některých komunikačních systémů nebo výstupní LCD displaye. Způsob, kterým jsou kódovány vstupní a výstupní signály, je většinou specificky určený daným řízeným systémem. [[Sass2010](#)]

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. [[Sass2010](#)].

Na obrázku ?? je zobrazeno názorné 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 a upraveno z [juan-fpgas])

### 3.0.2 Hardware Accelerated Applications

V mnoha aplikacích, nejen při řízení elektrických pohonů, je vyžadováno, aby výpočty nebo zpracování dat probíhalo vysokou rychlostí. Tento problém nemůže být většinou vyřešen 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 exponenciálně narůstající množství dat. 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* (hardwaerová akcelerace). [xilinx-accelerated-computing]

Princip hardwaerové 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 celkové aplikace. [xilinx-accelerated-computing]

Struktura, ve které je využíváno více fyzicky oddělených procesorových a hardwaerových akceleračních jednotek, se často nazývá heterogenní. [xilinx-accelerated-computing]

Hardwaerová akcelerace poskytuje rychlejší výpočty než CPU, protože využívá maximální paralelizace výpočtů. Klasické CPU však vykonává jednotlivé instrukce sériově. I v případě, že CPU má více jader a využívá více vláken, nemůže se specifické úrovni paralelismu při dané omezuje energetické náročnosti HW vyrovnat. Pro HW akceleraci je v mnoha oblastech využíváno několik druhů jednotek, které jsou optimální pro dané aplikace.

**Graphics Processing Units (GPUs)** jsou jednotky, které převážně slouží k akceleraci zpracování

vizuálních úloh. V době rychlého rozvoje elektroniky a SW je možné využítí GPUs v mnoha odvětví umělé inteligence (AI) či kreativních odvětví. GPUs jsou využívány v aplikacích, kde není kladen veliký důraz na nízkou odezvu (latenci). [\[xilinx-accelerated-computing\]](#)

**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 paměti, které by přenos dat zpomalovaly. [\[xilinx-accelerated-computing\]](#)

**Field Programmable Gate Arrays** (FPGAs) jsou jednotky, ve kterých není při výrobě pevně daná HW struktura. To umožňuje vytvoření, resp. naprogramování HW dle požadavků akcelerované aplikace. FPGAs mohou být využívány i při výpočtech v reálném čase matematických modelů elektrických strojů. Při realizaci této práce je pro akceleraci využíváno právě těchto programovatelných polí.

Porovnání časové náročnosti matematických výpočtů pro selektivní eliminaci harmonických složek v trakci pomocí CPU a GPU (graphics processing unit) je provedeno v [\[ieee-selective-harmonic-elimination-nvidia\]](#).

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

### 3.0.3 Výpočetní technika, mobilní zařízení a elektronika

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

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

Společnost Apple Inc. již téměř ve všech vlastních 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 značně zvýšit výkon a snížit energetickou náročnost zařízení spotřební elektroniky. [\[apple-explore-the-new-architecture-of-apple-silicon-macs\]](#)

## 4 Programovatelné hradlové pole – FPGA

### 4.1 Vývoj FPGA z PLD

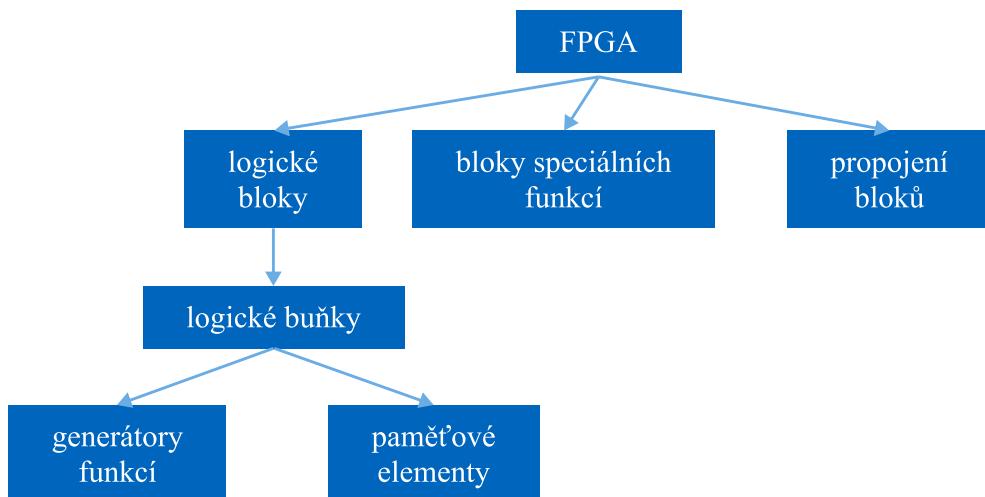
Programovatelné hradlové pole jsou zařízení, jejichž historický vývoj stojí na programovatelných logických zařízeních (programmable logic devices, PLD). První PLD fungovala na principu Booleových funkcí součtu násobení (sum of products). Tato zařízení obsahovala matici (proto se také nazývají programmable logic arrays, PLA) více vstupových bloků AND a OR. Programování požadované funkce probíhalo pomocí přerušování vstupů do jednotlivých logických bloků. Později byly do 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. [Sass2010]

### 4.2 Aktuální složení FPGA

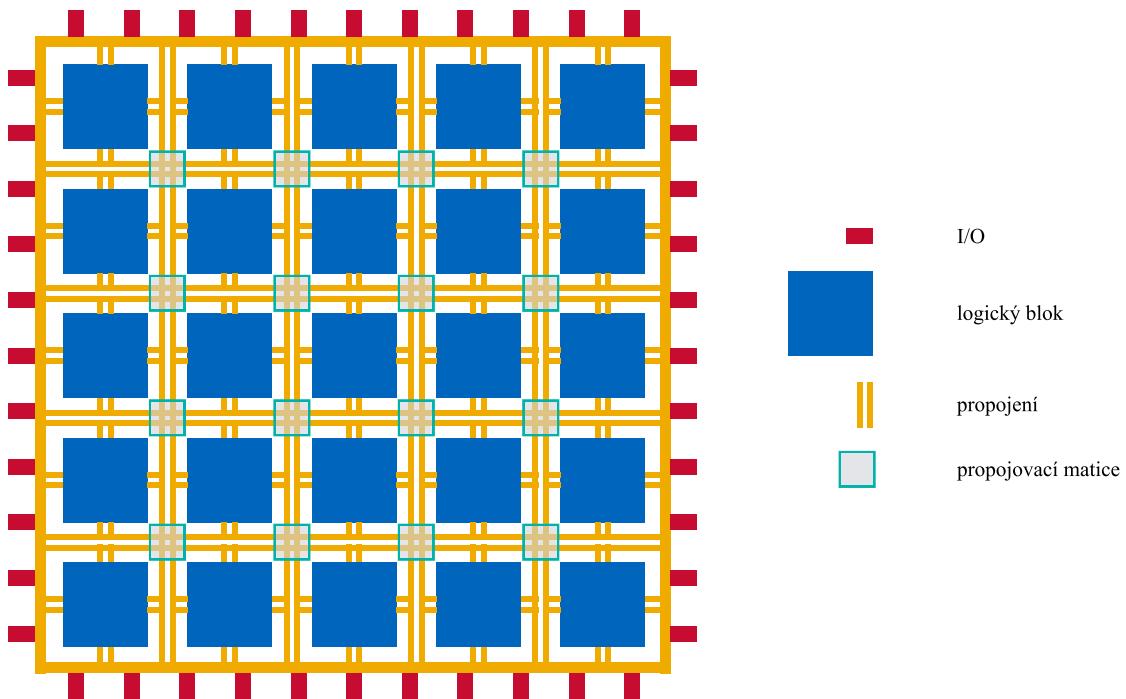
Moderní FPGA se skládají z 2D matice propojených programovatelných logických bloků, bloků speciálních funkcí a propojů vytvořených pomocí CMOS technologie. Po obvodě FPGA jsou rozmístěny vstupní a výstupní piny (I/O), připojené na zvláštní logické bloky. Použité logické bloky se skládají z mnoha buňek, které se skládají z generátorů funkcí a paměťových elementů. [Sass2010]

Na obr. ?? je možné pozorovat názorné 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 SOM 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 FPGA.

#### 4.2.1 Generátory funkcí

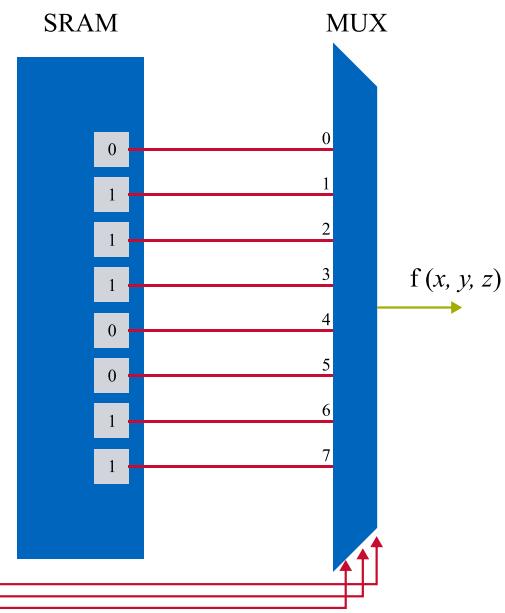
Oproti předchůdcům (PLD), které pro generování funkcí používaly logická hradla tvořená CMOS tranzistory, využívají FPGA generátory funkcí.

Logickou funkci je možné popsat pravdivostní tabulkou, která má určitý počet vstupů a odpovídající počet výstupů. Dle [Sass2010] 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 funkce 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. ???. Odpovídající realizace pomocí MUX a SRAM je zobrazena na obr. ???. Tato reprezentace se nazývá look-up table (LUT). Grafické znázornění inspirováno [Sass2010].

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.

Výhoda této reprezentace funkcí oproti logickým hradlům je, že doba zpoždění signálu (propagation delay) pro funkci je konstantní. Respektivně je konstantní, pokud funkci je možné realizovat jednou LUT. Pro realizaci obecné funkce je zapotřebí multiplexeru  $2^n \rightarrow 1$  a SRAM s počtem buněk  $2^n$ , kde  $n$  je počet vstupních proměnných dané funkce. [Sass2010]

#### 4.2.2 Paměťové elementy

Paměťové elementy jsou v FPGA realizovány pomocí D-klopových obvodů. Tyto obvody mohou při konfiguraci FPGA být nastaveny, že budou reagovat na nástupnou nebo sestupnou hranu časovacího signálu (clock, CLK) řídícího procesoru nebo na úroveň řídícího signálu (latch).[Sass2010]

Protože typ latch je citlivý na úroveň signálu, může být problematické dovést požadovaný signál na vstup klopového obvodu v požadovaném čase. Velmi často jsou proto paměťové členy konfigurovány jako D-klopové obvody reagující na hranu. Pokud je používán signál CLK vyšších frekvencí, je D-klopový obvod reagující na hranu snadněji schopný reagovat v požadovaném čase. [Sass2010]

Často jsou na vstup paměťových elementů připojeny výstupy multiplexerů generátorů funkcií. [Sass2010]

#### 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 vhodný pro jeho aplikaci. Pomocí logické buňky nebo skupiny logických buněk je již možné vytvářet plnohodnotnou kombinaci a sekvenční logiku.[Sass2010]

#### 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 jednotlivým

buňkami. Častá skutečnost je, že jednotlivé bloky mohou mít již předkonfigurovanou funkci, jako je např. sčítačka, dělička nebo násobička. [Sass2010]

#### 4.2.5 Propojení bloků

Propojení bloků je prováděno ke 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. [Sass2010] Na obr. ?? je prezentována 2D struktura pole. Ovšem pro zvětšení počtu LUTs, tudíž výpočetního výkonu, 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. [pang-beginning-fpga]

#### 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 informací 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. [Sass2010] [pang-beginning-fpga]

#### 4.2.7 Bloky speciálních funkcí

Aby došlo např. ke zvýšení 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 různorodé aplikace. [Sass2010]

**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 by 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. [Sass2010]

**DSP**, resp. digital signal processing bloky slouží ke zpracování digitálního signálu. V těchto blocích jsou implementované funkce AND, OR, NAND, NOT, násobičky a sčítáčky. Mají nízkou spotřebu. DSP bloky jsou často umístěny geograficky blízko bloků BRAM, které slouží jako „mezipaměti“ (buffer). [Sass2010]

**Procesor** implementovaný do struktury FPGA snižuje časové zpoždění při obsluhování FPGA. [Sass2010]

**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. [Sass2010]

**Multi-Gigabit Transcievers** slouží k přenosu dat takovým způsobem, aby došlo k minimalizaci vlivu ručení na přenášená data. Obecně obstarávají optimální serializaci a paralelizaci dat. [Sass2010]

Struktura FPGA, která je obsahuje všechny zdroje a funkcionality potřebné pro kompletní realizaci aplikace, se nazývá *platform FPGA*.

### 4.3 Programování

Ve skutečnosti není možné mluvit o programování FPGA jako o klasickém programování mikroprocesorů. Při tvorbě „programu“ pro FPGA dochází k vytváření struktury, jež bude následně v FPGA vytvořena. Ovšem z praktických důvodů se v praxi využívá pojmenování „programovat FPGA“.

### 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, VSIC HDL). 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 softwaerové) 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). [Sass2010]

Při použití HLS může vzniknout situace, že bude vytvořen algoritmus, který bude takovým způsobem komplexní, že ho nebude možné syntetizovat na FPGA. Oproti tomu při použití popisu pomocí structural/data flow circuits, je prakticky vždy algoritmus syntetizovatelný. [Sass2010]

Dalším negativním jevem je vytvoření neoptimalizovaného komplexního algoritmu, který se ve vyšším programovacím jazyce jeví jako jednoduchý, ale při překladu do HDL a následných kroků *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.

V praxi je k tvorbě algoritmů často využíváno vyšších programovacích jazyků a HLS, protože je tento přístup pro značný počet vývojářů SW srozumitelnější. Dalším častým přístupem v praxi je použití specializovaných SW jako je MATLAB™ a Simulink, které jsou schopny při použití odpovídajících balíčků přeložit vytvořený algoritmus do HDL, který je poté možné dále zpracovat a použít pro konfiguraci FPGA. Ovšem využití přístupu se SW MATLAB™ je třeba disponovat podporovaným HW, který disponuje dostatečným počtem 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 převáděna na logické členy AND, OR apod. [Sass2010]

Po vytvoření netlistu je nutné rozhodnout, jakým způsobem je možné a výhodné realizovat jednotlivé bloky v logických buňkách a LUT. Konečné sloučení 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.). [Sass2010]

Po mapování následuje proces umisťování (placement) při kterém je rozhodováno které z logických bloků budou realizovat FPGA členy, získané v kroku MAP. [Sass2010]

Bloků, které jsou umístěny ve struktuře FPGA je nutné spojit pomocí dostupných propojů na FPGA. Proces spojování a optimalizace propojů takovým způsobem, aby bylo minimalizováno časové zpoždění signálu, se nazývá *routing*. Obvykle se proces sloučuje s MAP do jedné fáze a nazývá se *place-and-*

*route* (PAR). [Sass2010]

Posledním krokem je vytvoření binárního souboru, nazývaného *bitstream*, který je poté „programováno“ 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 SRAM, ovládající příslušné propoje a matice. [Sass2010]



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 nízkou spotřebu energie oproti CPU nebo GPU. Ovšem oproti ASICs FPGA má stále značnější spotřebu, proto je podnikán 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 [[rovore-sphery-vs-shapes](#)], 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 on-line aplikace ale pro vykreslování nepohyblivých obrazů, které není nutné zobrazovat v reálném čase. [[wikipedia-ray-tracing](#)]

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. [[rovore-sphery-vs-shapes](#)].

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. [[grover-reduction-of-power-consumption](#)]

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. [[grover-reduction-of-power-consumption](#)]

## 4.5 Využití

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

### 4.5.1 Aplikace v nepohonářských odvětví

Díky univerzalitě PGAs je možné je využít v mnoha aplikacích různých odvětví. Stále se zvyšující požadavky na výpočetní výkon urychlují nasazování PGAs do provozů, kde jsou v současné době instalovány CPU nebo GPU.

Poptávka po dostupnosti FPGA způsobila vznik Cloud služeb, které nabízí FPGA výkon on-demand. Jedním z velkých 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 dostupný výpočetní výkon, který mohou v průběhu tvorby, debugingu či realizace aplikace měnit bez nutnosti pořizování výkonných a někdy drahých FPGA zařízení. Více o *Amazon EC2 F1 Instances* služby virtuálních FPGA je dostupné na [\[amazon-ec2-f1\]](#).

Existuje mnoho výpočetně náročných aplikací jako jsou např. výpočty finančních modelů pro ekonomiku, výpočty pro bioinformatiku, seismické modelování při hledání vzácných surovin apod. které je vhodné realizovat pomocí hardwaerového akcelerátoru. Více informací o těchto výpočetně náročných aplikacích je možné získat v [\[wim-high-performance-computing-using-fpgas\]](#).

Na akceleraci zpracování audiovizuálních děl je převážně určeno GPU. Ovšem pro aplikace, v nichž je vyžadováno zpracování obrazu v reálném čase s minimální spotřebou energie a nízkou hmotností aplikace, je často využíváno FPGA. Aplikace využití FPGA pro vozidla, která analyzují okolní prostor jsou popsány v [\[andina-advanced-features-and-industrial-applications-of-fpga\]](#). 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. [\[andina-advanced-features-and-industrial-applications-of-fpga\]](#)

Protože momentálním 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 zkoumání určitých kritických stavů těchto soustav by mohlo být ekonomicky i technicky nevýhodné. V tomto případě je vhodné vytvořit přesný matematický model jednotlivých analyzovaných součástí a nezbytné náročné výpočty akcelerovat pomocí FPGA. Na základě odezvy modelu je poté možné analyzovat stavy, které by v případě analýzy na reálném stroji mohly způsobit jeho destrukci či částečnou ztrátu funkčnosti. Proto se v průmyslu využívá Hardware-in-the-loop simulation (HILS), kdy je vytvořen požadovaný matematický model, který poskytuje elektrické signály do testovaného systému a na základě jeho reakce je možné vyhodnotit, díky matematickému modelu, jakým způsobem by se choval reálný modelovaný systém. [\[andina-advanced-features-and-industrial-applications-of-fpga\]](#), [\[mathworks-discovery-hil-simulation\]](#)

Kromě HIL 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 [\[naouar-fpga-based-current-controllers-for-ac-machine-drives\]](#). V dokumentu jsou popisovány tři realizace řízení, resp. regulace pohonu. Nejprve byla regulace realizována pomocí hystérezních on-off regulátorů, následně byly použity PI regulátory. Pomocí nich byl pohon regulován na základě měření a změny vektoru statorového proudu, resp. jeho složek  $\alpha\beta$  po aplikování Clarkové transformace. Jako poslení prezentovaný způsob autoři realizovali model ovládání synchronního motoru na základě prediktivních regulátorů. [\[naouar-fpga-based-current-controllers-for-ac-machine-drives\]](#)

Všechny prezentované způsoby regulace v [\[naouar-fpga-based-current-controllers-for-ac-machine-drives\]](#) byly před syntézou realizovány v prostředí MATLAB™ a Simulink. Tento způsob tvorby modelů a algoritmu je v praxi upřednostňován, protože umožňuje i expertům na řízení a regulaci pracovat na dané pro-

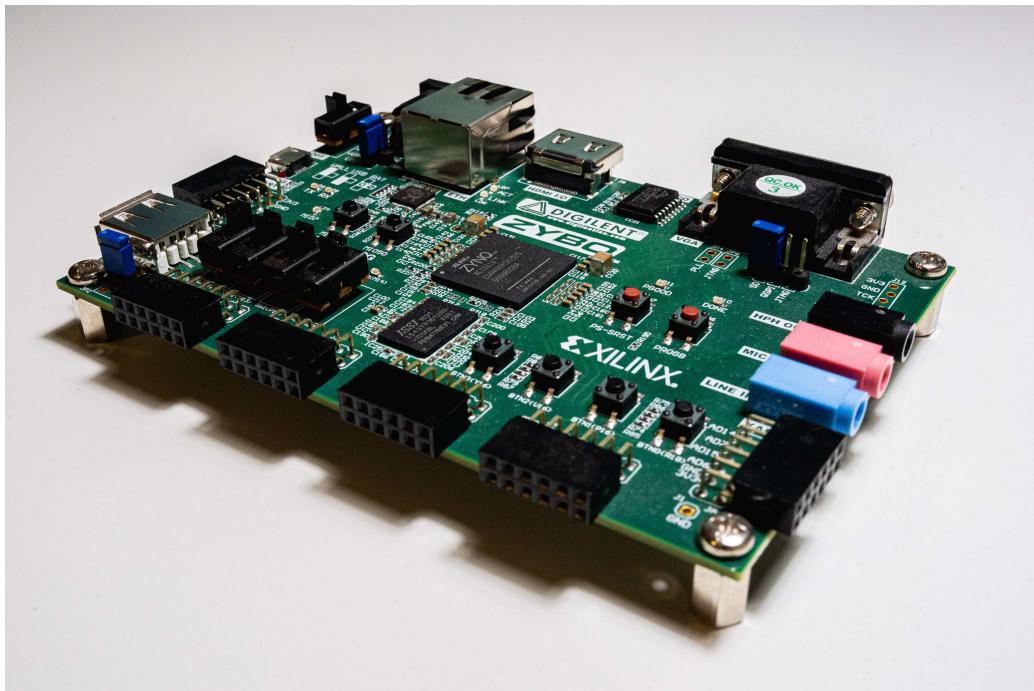
blematice bez znalostí mikroelektroniky, programování v HDL a způsobu fungování FPGA. Oproti tomu je třeba zvážit, jaké jsou požadavky na rychlosť, výkonnosť a optimalizované řízení aplikace a zdali použití předpřipravených knihoven a zjednodušených nástrojov nebude mít příliš značný vliv na rychlosť výpočtu a tudíž zpracování dat a řízení v reálném čase. [naouar-fpga-based-current-controllers-for-ac-machine-drives]

## 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 dokonce možné si sestavit vlastní HW s dostupných komponent takovým způsobem, aby vyhovoval požadované embedded aplikaci. Výhodné ovšem je využít již připravená řešení vývojových desek, které zjednoduší prvotní tvorbu aplikace.

V této práci byl realizován prvotní vývoj a seznámení s prostředím akcelerovaných aplikací na vývojové desce *Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board* od firmy Digilent. [[digilent-zybo-7000-docs](#)] 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 LUT, klopných obvodů a s rozsáhlější pamětí RAM. Bližší porovnání specifikací těchto desek je dostupné na [[digilent-zybo-compare](#)].

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 XC7Z010), který umožňuje pro vývoj aplikací použít SDK od firmy Xilinx. V tomto čipu je integrován dvou jádrový procesor ARM Cortex-9, který slouží jako pro řízení akcelerovaných aplikací na Xilinx FPGA sedmé série. Detailní schéma blokové architektury SoC

s označním sběrnic a komunikace jednotlivých částí čipu je zobrazené na obr. ??.

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 podpoře běhu hostitelského programu. 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 zakonponované 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 nař. 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.

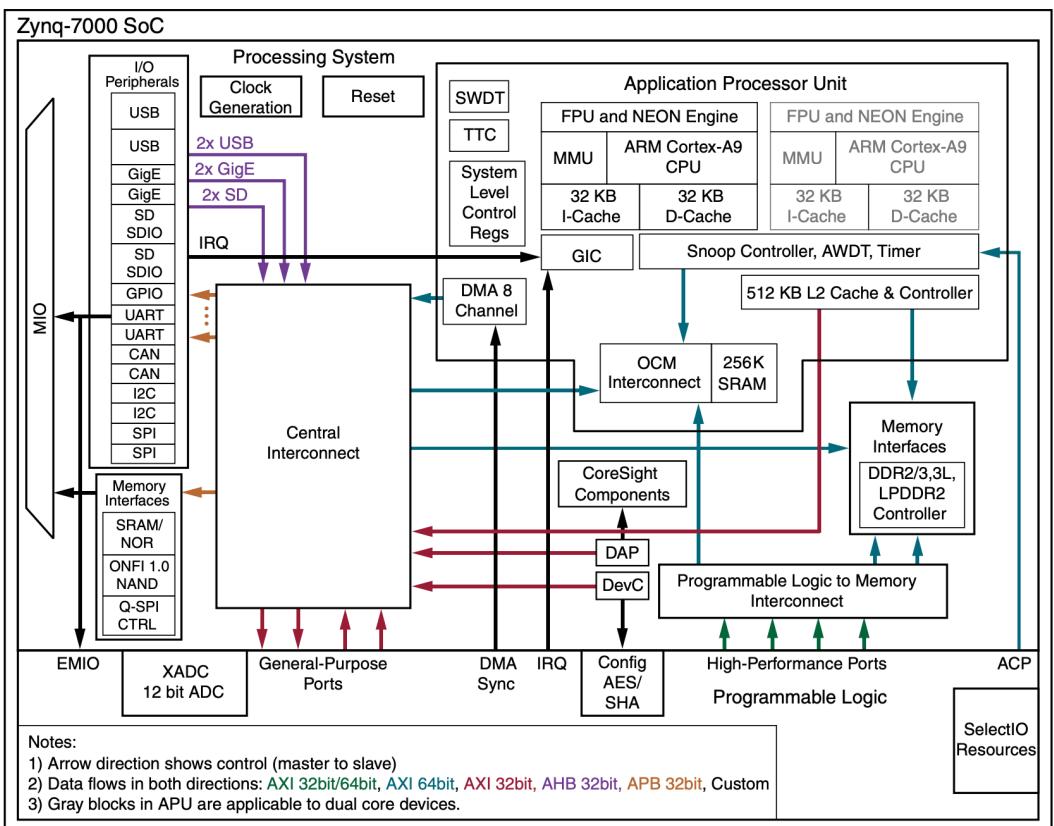
### Blok Interconnect

Blok Interconnect, resp. na obr. ?? 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 bloky jako např. blok zpracování digitálních signálů, řízení taktovacích hodin, analogově digitální převodník (ADC) apod.

Detailní technické specifikace, složení a parametry jmenovaných bloků a jsou uvedeny v [[xilinx-zynq-7000-technical](#)-

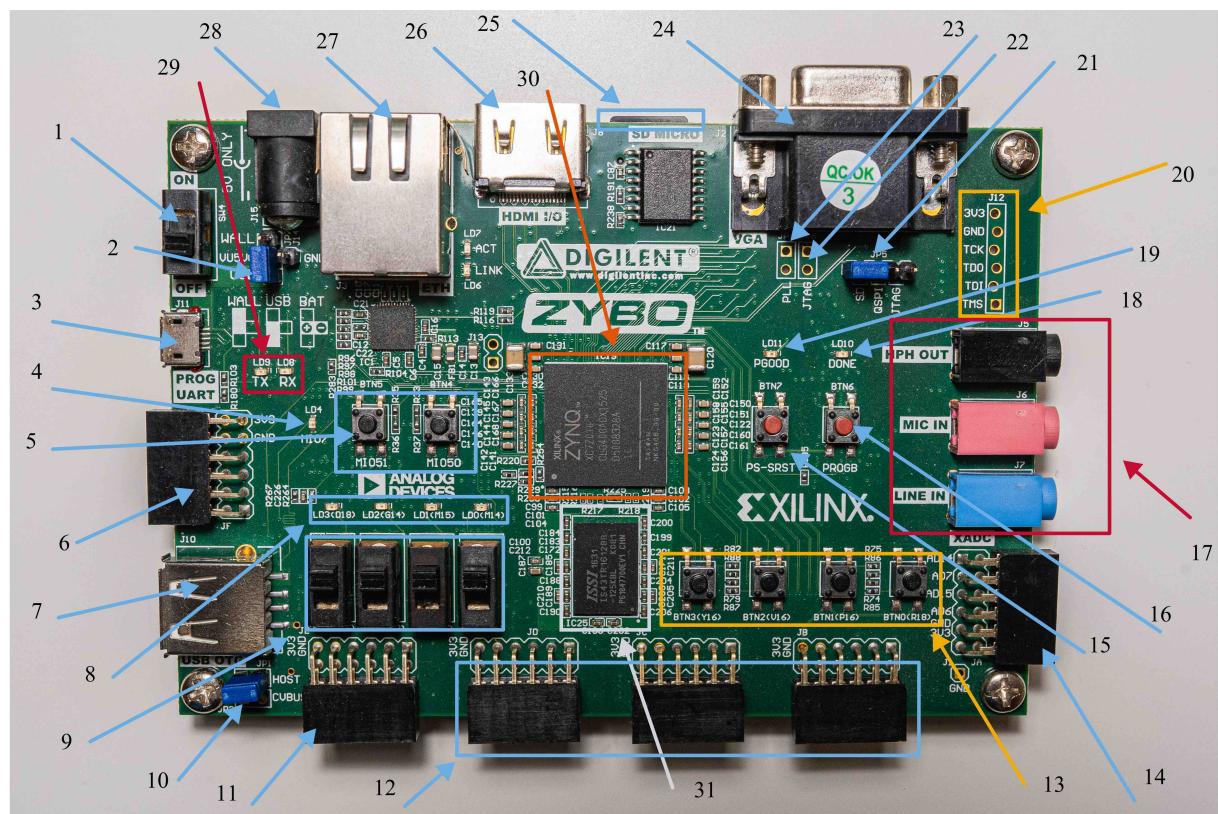


Obr. 5 - 2 Detailní schéma čipu Zynq-7000, umístěného na vývojové desce Digilent ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board. (převzato z [xilinx-zynq-7000-technical-reference-manual])

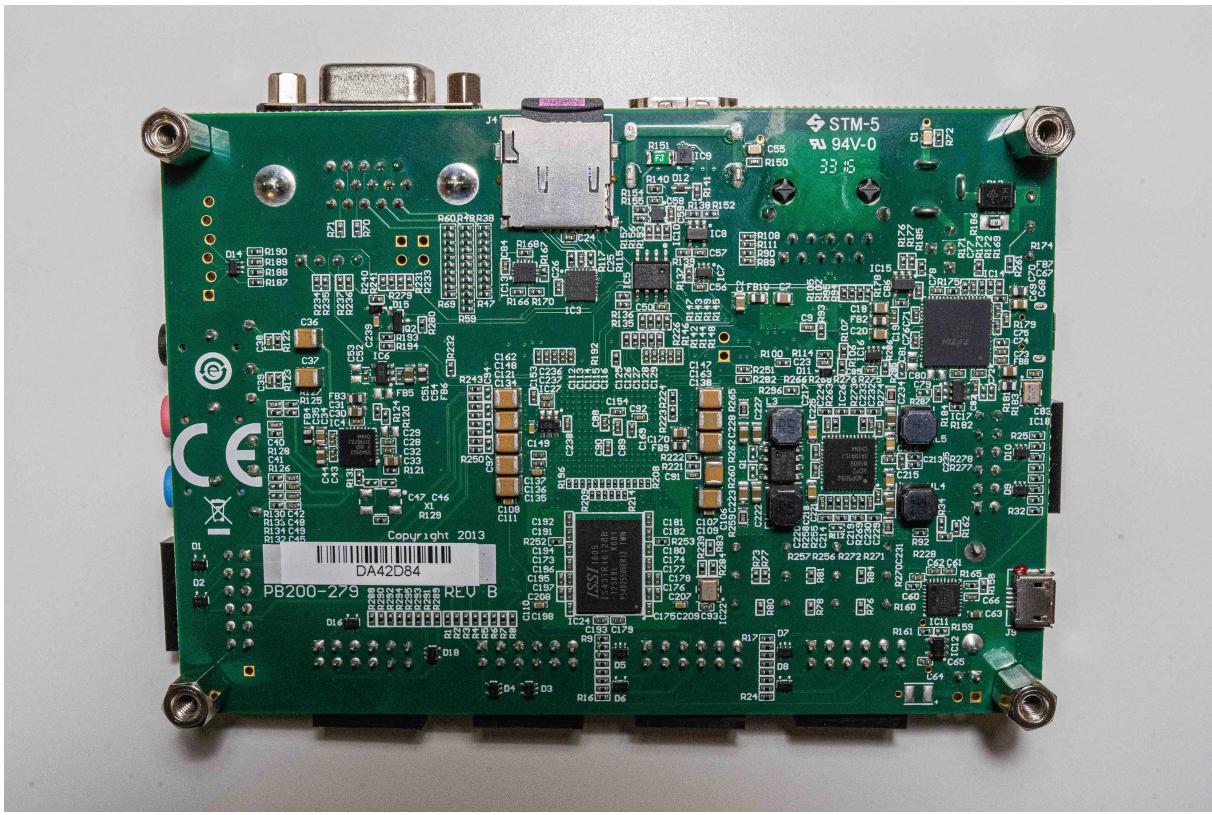
### 5.1.2 Uspořádání vývojové desky Zybo Zynq-7000

Na obr. ?? je zobrazen horní pohled na vývojovou desku, na které jsou vyznačeny významné části, kterým je vhodné věnovat pozornost. Číselné označení koresponduje s označením a vysvětlivkou v tabulce ??.

Pro úplnost je spodní strana desky zobrazena na obr. ??.



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 [\[digilent-zybo-reference-manual\]](#))*

| 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 displaye   |
| 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                        | srdce desky  |
| 31       | DDR2 Memory                            | RAM  |

## 6 Vývojová deska Xilinx Kria KR260

Deska od firmy Digilent, představená v *Vývojová deska Digilent Zybo*, je vhodná pouze pro prvotní seznámení s vytvářením akcelerovaných aplikací. Pro náročnější aplikace, které při potřebují při využití většího množství LUTs nebylo v této práci možné Zybo použít. Vývojová deska Kria KR260 disponuje dostatečným množstvím LUTs a díky svým moderním komponentám a prvkům, může efektivně sloužit k vytváření náročnějších aplikací.

Hlavní částí vývojové desky KR260 je „modul“ *Kria K26 System-on-Module*. Tedy 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 také po ukončení vytvářeného návrhu CC) je možné zakoupit pro aplikaci v průmyslu samotný modul ve vhodné variantě. V této práci je ovšem využíván standardní vývojový „Starter kit“ s deskou KR260, jejíž komponenty je vhodné představit.

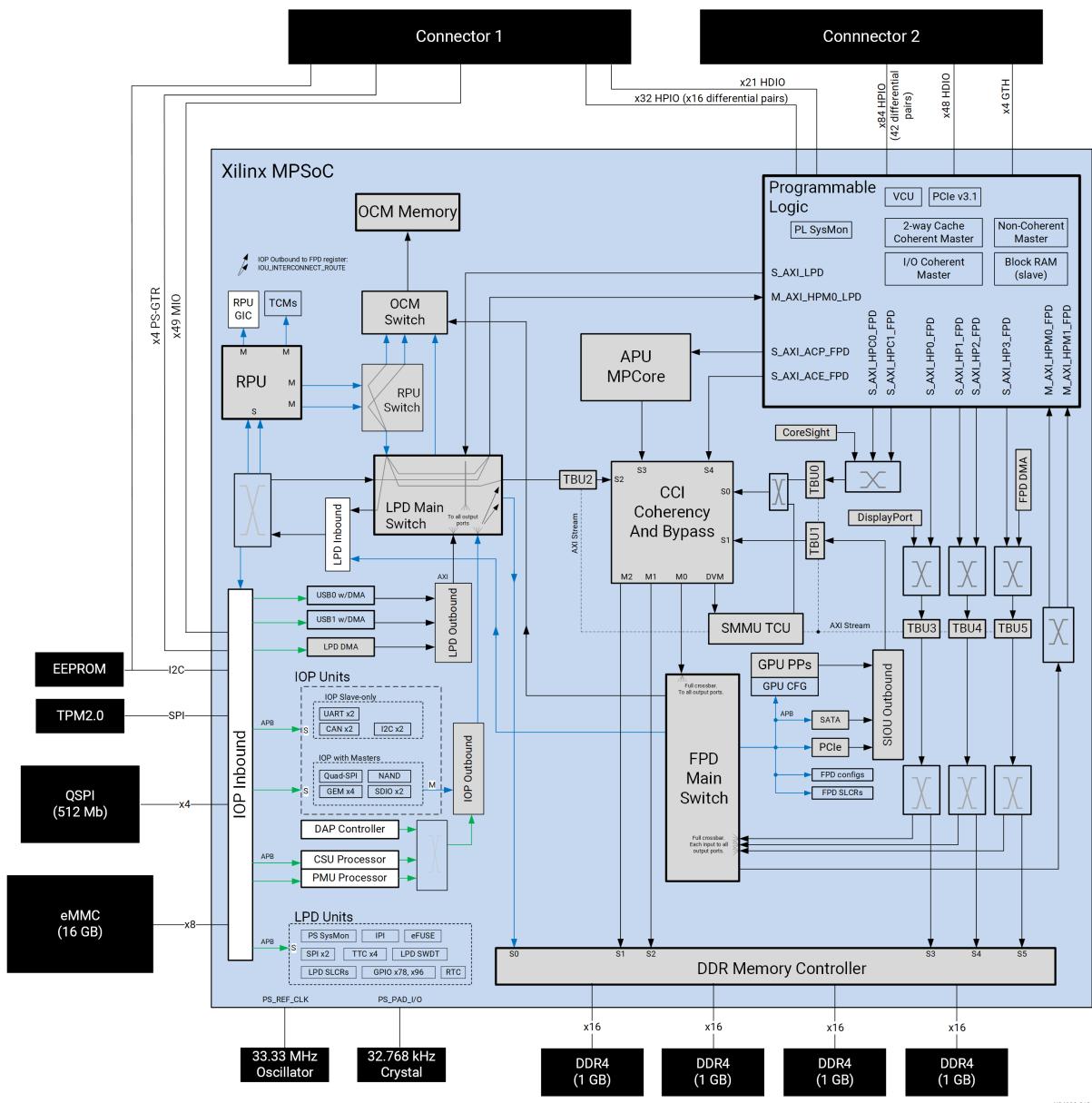


Obr. 6 - I Vývojová deska Xilinx Kria KR260 – boční pohled.

### 6.1 Základní přehled

#### 6.1.1 CPU a FPGA čip

Strukturu SOM je možné popsat jako modernější a rozmanitější vylepšení SoC. Opět se vše struktury nachází hlavní základní bloky pro PS a PL. Ukázkový blokový diagram struktury udávané výrobcem je na obr. ??.



Obr. 6 - 2 Blokový diagram K26 SOM Kria. [kria-k26-som-ds]

Největšími rozdíly mezi použitými deskami Zybo a Kria je např. velikost operační paměti, počet jader a taktovací frekvence procesorů v PS, počet LUTs nebo logických buněk v FPGA (PL). Úplné specifikace pro K26 SOM je možné nalézt v [[\[kria-k26-som-ds\]](#)].

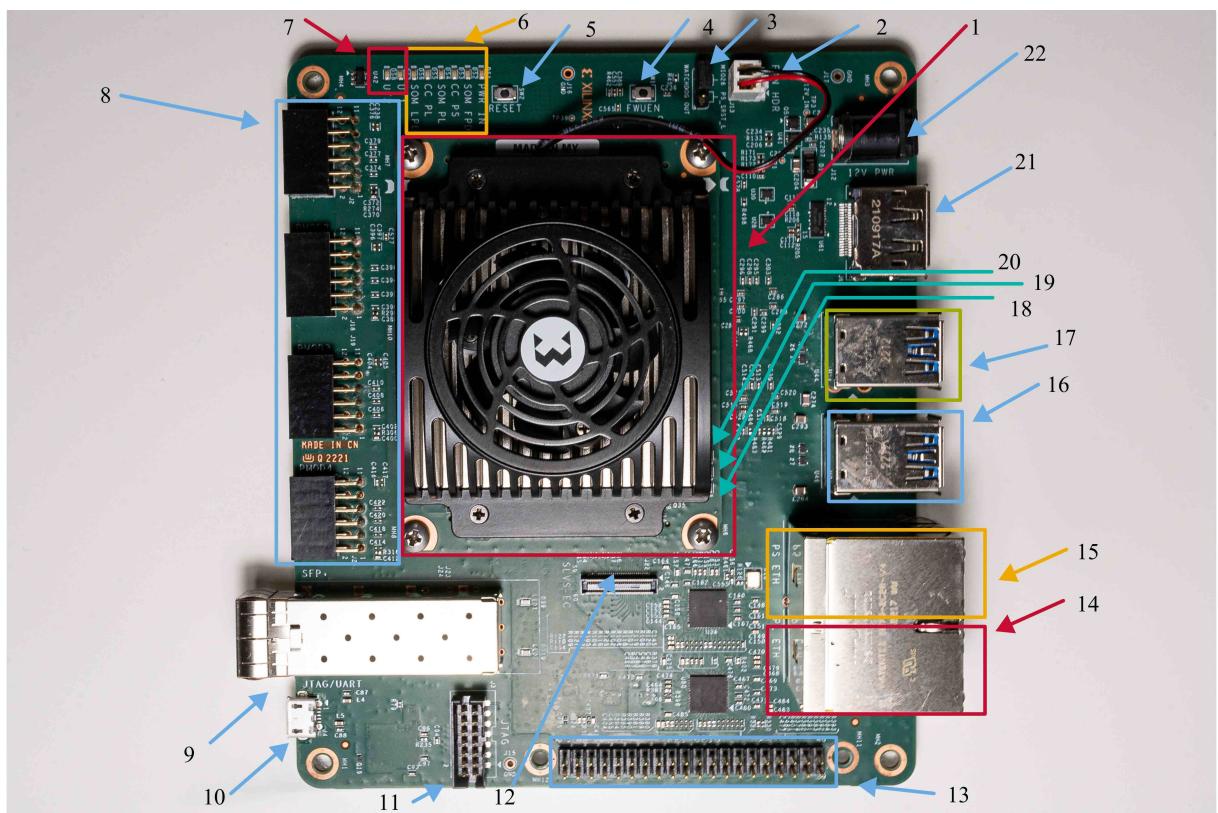
## 6.2 Uspořádání vývojové desky

Na obr. ?? je zobrazen horní pohled na vývojovou desku, na které jsou vyznačeny významné části, kterým je vhodné věnovat pozornost. Číselné označení koresponduje s označením a vysvětlivkou v tabulce ??.

Na spodní straně desky je umístěn slot pro SD kartu, na kterou je umisťován operační systém pro PS. Pro úplnost je spodní strana desky zobrazena na obr. ??.

*Tab. 6 - 1 Popis označených komponent na vývojové desce Xilinx Kria KR260. (informace a značení převzaty z [[\[kria-kr260-robotics-starter-kit-user-guide\]](#)])*

| 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, jsou 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 PL          |
| 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*. Nadřazeným parametrem dvou hlavních variant jsou varianty 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 zabudované *crypto-accelerator* bloky. Encryption Enabled varianty jsou v některých zemích zakázané a proto je důležité při výběru zboží dbát pokynům prodejce.

Hlavní rozdíly *Commercial* a *Industrial* variant jsou uvedeny v tab. ??.

Tab. 6 - 2 Porovnání hlavních parametrů Kria K26 SOM Commercial a Industrial. (informace a značení převzaty z [\[kria-k26-som-product-brief\]](#))

| 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 disponuje dle informací výrobce SOM, který není určen pro nasazení do aplikací (non-production) v teplotní třídě *Commercial*. [\[kria-k26-som-ds\]](#)

## 7 Porovnání představených SoC/SoM platforem pro řízení elektrických pohonů

V předchozích kapitolách byly představeny dvě smysluplné, komerčně dostupné platformy, které je možné využít pro různorodé aplikace. Výběr byl zaměřen na SoC a SoM umístěných na vývojových deskách, které je možné využít pro vývoj požadované aplikace. Často po vývoji aplikace následuje uvědomení, jakými periferiemi a vlastnostmi by měla architektura řídící soustavy disponovat. Poté je možné pro velko produkci dané aplikace začít vyvíjet vlastní integraci čipu/modulu na Printed Circuit Board (PCB).

### 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 např. drivery spínacích polovodičových prvků či výkonových polovodičových můstků. PMOD konektorů využívá mnoho komerčně dostupných prvků jako jsou senzory, H můstky nebo také ADC/DAC převodníky.

Dalším důležitým faktorem je připojení Ethernet. Obě desky disponují alespoň jedním konektorem. Xilinx Kria KR260 obsahuje 4 konektory, přičemž dva jsou připojeny přímo do PS a dva do PL. Digilent Zybo disponuje pouze jedním konektorem.

Pro připojení periferií nebo externích datových úložišť je možné využít konektor USB. Protože Digilent Zybo Z7 je staršího data vydání, disponuje pouze USB 2.0, zatímco Xilinx Kria KR260 disponuje připojením 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 poté disponuje konektory pro výstup reproduktorů či vstup mikrofonu. Deska KR260 těmito konektory nedisponuje.

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

Protože je K26 SOM zaměřen také na AI je možnost připojit k desce kamery pomocí konektoru SLVS-EC.

Posledním významným konektorem na CC Xilinx Kria je SPF 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*, obsahuje PS použité vývojové deska Digilent Zybo Zynq-7000 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 také doplněn dvoujádrovým real-time dvoujádrovým procesorem Arm Cortex-R5F MPCore s taktovací frekvencí až 600 MHz. [\[digilent-zybo-reference-manual\]](#), [\[kria-k26-som-ds\]](#)

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 obsahují novější verze Zynq

čipu, který nabízí také 17 600 LUTs (Zybo Z7-10) nebo 53 200 LUTs (Zybo Z7-20). [[digilent-zybo-reference-manual](#)], [[kria-k26-som-ds](#)]

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

Vlivem omezených zdrojů při realizaci této práce bylo možné i přes optimalizaci C++ kódu (pro dodržení load-compute-store modelu programování [[vitis-unified-software-platform-documentation-2022](#)])) akcelerované aplikace (kernel) pro FPGA možné umístit do PL v Digilent Zybo Z7 pouze matematický I-n model asynchronního motoru, jehož výsledkem byl transformační úhel, složky magnetického toku rotoru  $\psi_2^{\alpha\beta}$  a velikost magnetického toku rotoru  $|\psi_2|$ . Oproti tomu při využití Kria K26 SOM je možné využít PL pro výpočet I-n modelu stroje, zjednodušeného modelu asynchronního motoru i regulačního zásahu.

## 7.3 Developer Experience

V moderní době, kdy je kladen značný důraz na rychlosť vývoje aplikace, je důležitým hodnotícím faktorem developer experience. Tudíž jak je systém konfigurace a vytváření aplikací přívětivý pro vývojáře. V době vydání Digilent Zybo Z7 byl používán postup vytvoření operačního systému Petalinux již s pevně daným Device Tree (DT), který byl možné měnit pomocí celkové 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 PL část, 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 *Petalinux*.

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 Vivado již „board files“ zakomponovány. 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 [[kria-kr260-starter-kit-cc-schematics](#)] a [[kria-k26-som-xdc](#)].

## 7.4 Aplikace a operační systém

Aplikace pro Digilent Zybo je možné vytvářet jako *Bare Metal / Standalone* nebo aplikace pro operační systém *Petalinux*. Pro Xilinx Kria je možné využít *Bare Metal / Standalone*, *PetaLinux* a také distribuci operačního systému *Linux Ubuntu*. Protože obě využívané vývojové desky využívají čipu od firmy Xilinx, Inc., je podpora dostupná pro oba čipy relativně srovnatelná. Výrobce na stránkách Wiki podpory zmiňuje, že poskytuje podporu převážně ve formě veřejného fóra na adrese [support.xilinx.com](http://support.xilinx.com). Podpora bude dostupná pro dvě poslední major verze softwarových nástrojů, které jsou součástí *PetaLinux* a *Xilinx SDK*. [[xilinx-wiki-atlassian-embedded-sw-support](#)]

Protože je rodina Kria SOMs modernější než Digilent Zybo, výrobce vytvořil ukázkové akcelerované aplikace, které je možné při využívání operačního systému *Ubuntu* přímo stáhnout z Kria App Store. [\[xilinx-appstore-for-kria-soms\]](#) Oba čipy 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 [\[wiki-linux-foundation-real-time-linux\]](#) nebo z omezené podpory pomocí fóra. Více informací o postupu realizace patche *PetaLinux* je v sekci *RealTime Linux Patch*.

## 8 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 využito matematického modelu asynchronního motoru. Asynchronní motor bude modelován pro řízení pomocí  $I$ - $n$  modelu a pro simulační účely pomocí zjednodušeného modelu, zanedbávající některé jevy.

Pokud FPGA obsahuje dostatečné množství zdrojů, je možné realizovat akcelerovaný výpočet „kompletního“ matematického modelu, ve kterém např. dochází k simulovanému generování vstupního napětí, jehož popis pomocí rovnic je představen v části *Matematický popis „kompletního“ modelu stroje*. V části *PL a PS* 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 apod.

### 8.1 Představení stroje

### 8.2 Matematický popis „kompletního“ modelu stroje

Tab. 8 - 2 Změřené parametry stroje.

Tab. 8 - 1 Štítkové údaje stroje.

|                   |                         |
|-------------------|-------------------------|
| $P_n$             | 12 kW                   |
| $U_n$             | 380 V                   |
| $I_n$             | 22 A                    |
| $n_n$             | $1460 \text{ min}^{-1}$ |
| $f_n$             | 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 <sup>2</sup> |

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,  $f_n$  (Hz) je jmenovitá napájecí frekvence stroje,  $\cos(\varphi_n)$  (-) je jmenovitý účinník stroje,  $n_n$  ( $\text{min}^{-1}$ ) jsou jmenovité otáčky stroje,  $p_p$  (-) je počet polpáru stroje,  $R_1$  (Ω), resp.  $R_2$  (Ω) je statorový, resp. rotorový odpor,  $L_{1\sigma}$  (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.

V případě tvorby modelu, je využit model založen na výpočtu složek vektorů statorového proudu  $i_1$  a rotorového toku  $\psi_2$  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.

$$\frac{d}{dt} \begin{bmatrix} i_{1\alpha} \\ i_{1\beta} \\ \psi_{2\alpha} \\ \psi_{2\beta} \end{bmatrix} = \begin{bmatrix} -\frac{R_2 L_m^2 + L_2^2 R_1}{\sigma L_1 L_2^2} & 0 & \frac{L_m R_2}{\sigma L_1 L_2^2} & \frac{L_m}{\sigma L_1 L_2} \omega \\ 0 & -\frac{R_2 L_m^2 + L_2^2 R_1}{\sigma L_1 L_2^2} & -\frac{L_m}{\sigma L_1 L_2} \omega & \frac{L_m R_2}{\sigma L_1 L_2^2} \\ \frac{L_m R_2}{L_2} & 0 & -\frac{R_2}{L_2} & -\omega \\ 0 & \frac{L_m R_2}{L_2} & \omega & -\frac{R_2}{L_2} \end{bmatrix} \begin{bmatrix} i_{1\alpha} \\ i_{1\beta} \\ \psi_{2\alpha} \\ \psi_{2\beta} \end{bmatrix} + \begin{bmatrix} \frac{1}{\sigma L_1} & 0 \\ 0 & \frac{1}{\sigma L_1} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_{1\alpha} \\ u_{1\beta} \end{bmatrix}. \quad (8-1)$$

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

$$M = \frac{3}{2} p_p \frac{L_m}{L_2} (\psi_{2\alpha} i_{1\beta} - \psi_{2\beta} i_{1\alpha}), \quad (8-2)$$

$$M - M_z = J \frac{d\Omega}{dt}, \quad (8-3)$$

$$\omega = p_p \Omega, \quad (8-4)$$

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  $\underline{i}_1$  (A),  $\psi_{2\alpha}$  (Wb) a  $\psi_{2\beta}$  (Wb) jsou složky vektoru rotorového magnetického toku  $\underline{\psi}_2$  (Wb),  $u_{1\alpha}$  (V) a  $u_{1\beta}$  (V) jsou složky statorového napětí  $\underline{u}_1$  (V),  $p_p$  (-) je počet polpárů stroje,  $\omega$  ( $s^{-1}$ ) je elektrická úhlová rychlosť hřídele,  $\Omega$  ( $s^{-1}$ ) je mechanická úhlová rychlosť hřídele,  $M$  je vnitřní elektromechanický moment stroje a  $M_z$  (Nm) je moment zátěžný.

### 8.3 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 např. proudově-otáčkového, resp. *I-n* modelu a regulační procesy realizovat v PS.

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

$$\underline{u}_1^k = R_1 \underline{i}_1^k + \frac{d\underline{\psi}_1^k}{dt} + j\omega_k \underline{\psi}_1^k, \quad (8-5)$$

$$\underline{u}_2^k = R_2 \underline{i}_2^k + \frac{d\underline{\psi}_2^k}{dt} + j(\omega_k - \omega) \underline{\psi}_2^k, \quad (8-6)$$

$$\underline{\psi}_1^k = L_1 \underline{i}_1^k + L_m \underline{i}_2^k, \quad (8-7)$$

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

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,  $\omega$  ( $s^{-1}$ ) je úhlová rychlosť otáčení rotoru,  $\omega_k$  ( $s^{-1}$ ) je úhlová rychlosť otáčení použitého souřadnicového systému,  $R_1$  ( $\Omega$ ), resp.  $R_2$  ( $\Omega$ ) 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 tudíž model bude odvozován v souřadnicovém systému spojeným se statorem (souřadnicový systém  $\alpha\beta$ ). Po vzdelení daného souřadnicového systému pro soustavu rovnic platí

$$\underline{u_1^{\alpha\beta}} = R_1 \underline{i_1^{\alpha\beta}} + \frac{d\underline{\psi_1^{\alpha\beta}}}{dt}, \quad (8 - 9)$$

$$\underline{u_2^{\alpha\beta}} = R_2 \underline{i_2^{\alpha\beta}} + \frac{d\underline{\psi_2^{\alpha\beta}}}{dt} - j\omega \underline{\psi_2^{\alpha\beta}}, \quad (8 - 10)$$

$$\underline{\psi_1^{\alpha\beta}} = L_1 \underline{i_1^{\alpha\beta}} + L_m \underline{i_2^{\alpha\beta}}, \quad (8 - 11)$$

$$\underline{\psi_2^{\alpha\beta}} = L_2 \underline{i_2^{\alpha\beta}} + L_m \underline{i_1^{\alpha\beta}}. \quad (8 - 12)$$

V případě řízení asynchronního motoru s kotvou nakrátko orientovaného na rotorový tok je dále z rovnice ?? vyjádřen prostorový vektor  $\underline{i_2^{\alpha\beta}}$ , který je dále dosazen do upravené rovnice ??, u které je přepo-kládáno, že  $\underline{u_2^{\alpha\beta}} = 0$ . Výsledná diferenciální rovnice pro prostorový vektor rotorového magnetického toku je

$$\frac{d\underline{\psi_2^{\alpha\beta}}}{dt} = \frac{R_2}{L_2} L_m \underline{i_1^{\alpha\beta}} + j\omega \underline{\psi_2^{\alpha\beta}} - \frac{R_2}{L_2} \underline{\psi_2^{\alpha\beta}}. \quad (8 - 13)$$

Rozepsáním představené diferenciální rovnice do reálné a imaginární složky vznikne soustava dife-renciálních rovnic, kterou je třeba řešit.

$$\begin{aligned} \frac{d\underline{\psi_{2\alpha}}}{dt} &= \frac{L_m R_2}{L_2} \underline{i_{1\alpha}} - \frac{R_2}{L_2} \underline{\psi_{2\alpha}} - \omega \underline{\psi_{2\beta}}, \\ \frac{d\underline{\psi_{2\beta}}}{dt} &= \frac{L_m R_2}{L_2} \underline{i_{1\beta}} - \frac{R_2}{L_2} \underline{\psi_{2\beta}} + \omega \underline{\psi_{2\alpha}}. \end{aligned} \quad (8 - 14)$$

## 9 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 na 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 v 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 SOM.

Veškerý používaný SW v této práci od firmy Xilinx je po registraci volně dostupný ke stažení na [\[xilinx-downloads\]](#).

### 9.1 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 pro FPGA může sloužit i jako jediný vývojářský nástroj.

Se znalostí VHDL je možné ve Vivado vytvářet požadované HW konstrukce, ovšem tvorba těchto konstrukcí ve Vivado je relativně náročnou záležitostí není předmětem této práce, ale je nevyhnutelnou součástí výzkumu využití těchto plaforem pro řízení elektrických pohonů.

Xilinx Vivado je součástí instalačního balíčku *Xilinx Unified Installer*, dostupného z [\[xilinx-downloads\]](#). Součástí balíčku verze 2022.2 SFD je také 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.

### 9.2 Xilinx Vitis

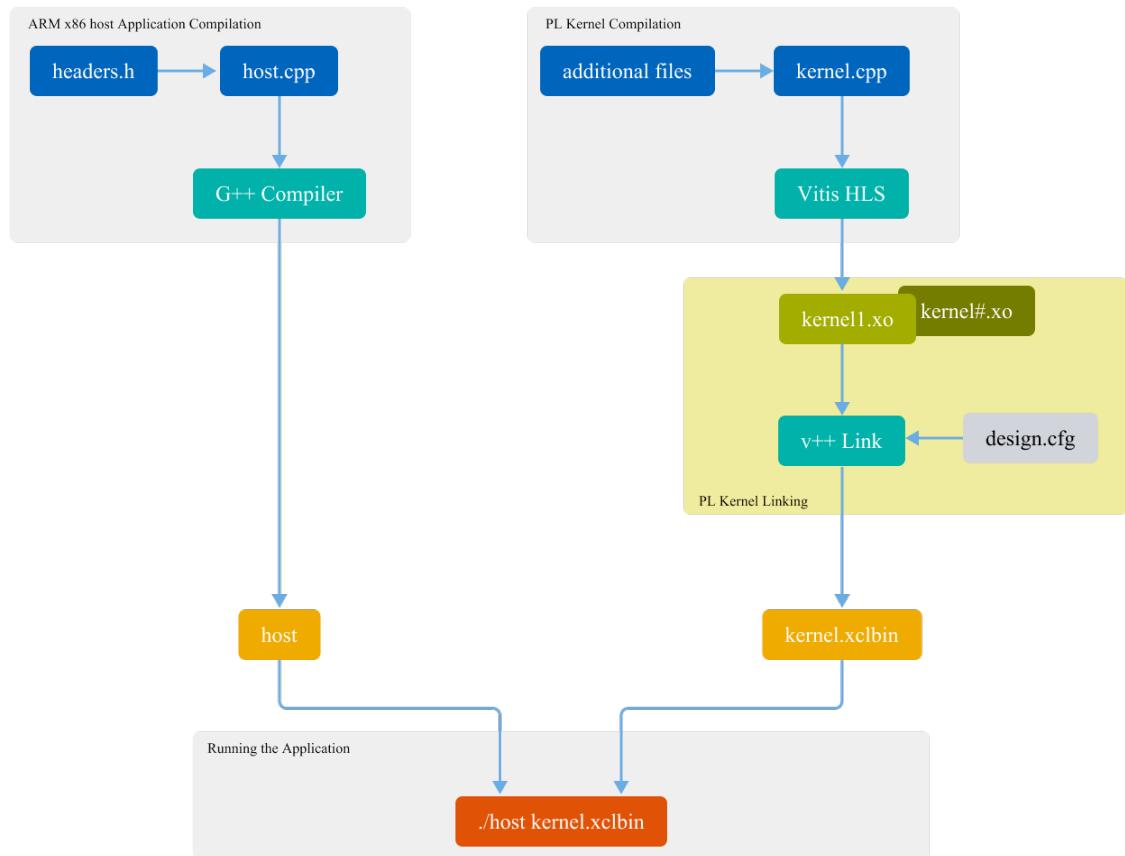
Xilinx Vitis je nástroj, který slouží k vytváření akcelerovaných aplikací na zařízení firmy Xilinx. Tento nástroj obsahuje základní vrstvu s názvem Xilinx Vitis HLS, která slouží jako jádro převodu vytvářených aplikací v C, C++, OpenCL do RTL. V programu Xilinx Vitis bude vytvářena největší část aplikace, proto je vhodné nastinit 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), umístěným v PL v FPGA. Blok *ARM x86 host Application Compilation* naznačuje, jakým způsobem je vytvářena aplikace pro host procesor. [\[vitis-unified-software-platform-documentation-2022\]](#)

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.[\[vitis-unified-software-platform-documentation-2022\]](#)

Na obr. ?? je blokově znázorněn postup tvorby spustitelné aplikace v programu Vitis. Tento diagram předpokládá, že již byla vytvořena HW platforma ve Vivado a PetaLinux systém.

Produkty větve 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 kompletně vykonáván v prostředí PetaLinux v simulátoru (QEMU) nebo na fyzickém zařízení (vývojová deska).



Obr. 9 - 1 Blokový diagram tvorby spustitelné aplikace v prostředí Vitis. (převzato z [\[vitis-unified-software-platform-documentation-2022\]](#), upraveno)

### 9.3 PetaLinux Tools

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

PetaLinux poskytuje distribuci systému Linux, který uživatel před tvorbou aplikace pomocí Xilinx Vitis využívá k tomu, aby vytvořil operační systém, na kterém bude možné spuštět dané akcelerované aplikace. Tento systém je možné nakonfigurovat dle požadavků aplikace. Při tvoření 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.). [\[xilinx-petalinux\]](#)

PetaLinux Tools mohou sloužit k debuggingu a virtualizaci systému pomocí QEMU. 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 build. Tvorba PetaLinux systému je časově náročný postup, u něhož je problematický debugging.

Pro funkční instalaci PetaLinux Tools je nutné disponovat nainstalovanými správnými verzemi systémových a aplikačních balíků, které jsou nutnou prerekvizitou tvorby PetaLinux pro PS. Požadavky na balíky je možné nalézt při nahlédnutí do dokumentace [[petalinux-tools-documentation-2022](#)] (UG1144) dané instalované verze PetaLinux Tools. V sekci *Installation Requirements* se nachází odkaz označený *PetaLinux <version> Release Notes*, který ve spodní části obsahuje stáhnutelný soubor *<version>\_PetaLinux\_Package\_List.xlsx*, který obsahuje požadované balíky a jejich verze. Bez použití podporovaných balíků by nepracoval nástroj PetaLinux Tools správně.

## 9.4 RealTime Linux Patch

Protože operační systém Linux nebyl původně vytvářen 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. Proto se objevila snaha upravit tento systém takovým způsobem, aby jej bylo možné využívat v real time systémech.

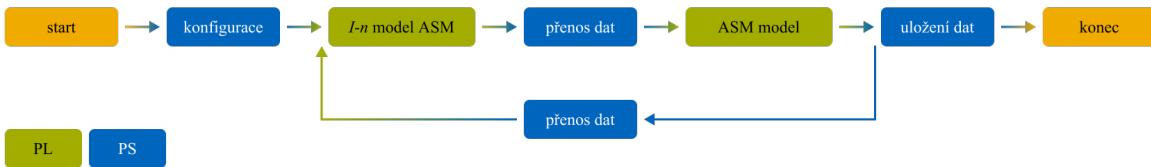
Real time systémy je obvykle možné rozdělit do jednotlivých úrovní podle časových požadavků systému v reálném čase na:

- **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 výpočtů, 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 [[the-real-time-linux-kernel-survey-on-preempt-rt](#)] jsou představeny původní přístupy, kdy pro dodržení časových omezení a tzv. „preemptibility“ (přerušitelnosti vykonávaného vlákna) byl využit *coker-nel*.

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 přidávaná do systému další vrstva jádra, ale původní jádro je upravováno. Úpravy spočívají ve změně některých přístupů funkčnosti jádra a přerušení. Tento patch se obecně nazývá *PREEMPT\_RT* a o začlenění jeho principů do mainline kernelu je dlouhodobě usilováno. [[the-real-time-linux-kernel-survey-on-preempt-rt](#)]

Popis state-of-art *PREEMPT\_RT* je popsán v [[the-real-time-linux-kernel-survey-on-preempt-rt](#)]. V této 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. ??, dojde při opakováném 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 opakováném spuštění aplikace je možné získat validní výsledky, ovšem četnost, kdy dochází k získání nevalidních výsledků, je značně vysoká.



Obr. 9 - 2 Graf prováděné simulace při testování PREEMPT\_RT Linux Patch.

#### 9.4.1 Postup aplikace PREEMPT\_RT patch

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

Pro bezproblémové aplikování patche je vhodné nejdříve vytvořit *build* PetaLinux systému bez aplikovanání patch souboru s minimální konfigurací a 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 ??, kde <petalinux-project> je označení pro kořenový (root) adresář PetaLinux projektu, který je tvořen.

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

Kód 9 - 1 Cesta Makefile souboru, ze kterého je možné získat označení verze jádra systému PetaLinux.

Protože je zmiňovaný **Makefile** soubor rozměrný a pro určení verze kernelu je signifikantní pouze jeho úvodní část, je v kódu č. ?? 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 9 - 2 Významná část Makefile souboru pro určení verze jádra PetaLinux.

Z kódu ?? je možné vyčíst, že je třeba využít patch pro verzi 5.15.36. Pokud není možné informaci ohledně verze jádra linuxu dohledat, 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 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 mít informaci o umístění daného souboru, proto je vyžadováno aby do souboru <petalinux-project>/**project-spec/meta-user/recipes-kernel/linux/linux-xlnx\_% .bbappend** zapsat na poslední řádek 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 ???. Jak je vidět, soubor obsahuje informace o různých konfiguračních souborech pro tvorbu jádra Linux systému. Šestý řádek označuje uživatelskou konfiguraci, pomocí data, kdy byla vytvořena.

```
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 9 - 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-confic -c kernel` pro konfiguraci jádra. Po otevření konfigurační nabídky kernelu je nutné provést změny, vyznačené v ???.

```
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 9 - 4 Úpravy v konfiguraci jádra pro RT patch.

Značení:

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

Po konfiguraci je již možné pokračovat klasickým způsobem build procesu popsaným v části *Tvorba PetaLinux*.

Představený postup čerpá informace o aplikování patch souboru z [[hackster-real-time-optimization-in-petalinux-trenz-electronic-wiki-how-to-install-the-linux-rt](#)] a z experimentálního zjištění autora.

## 9.5 Programovací prostředí – operační systém Linux

Pro práci s představenými nástroji Xilinx Vivado, Xilinx Vitis a PetaLinux Tools je nutné využívat podporovaných operačních systémů, které umožňují správnou funkci využitých nástrojů.

Jednotlivé požadavky na operační systémy je možné nalézt na stránkách dokumentace <https://docs.xilinx.com>. Pro nejnovější verze v době zpracování této práce jsou požadavky pro Xilinx Vivado dostupné v [[xilinx-vivado-design-suite-user-guide-2022](#)]. Požadavky na operační systém pro Xilinx Vitis v [[vitis-unified-software-platform-documentation-2022](#)]. Pro využívání a tvorbu PetaLinux Tools je třeba dodržet systémové požadavky uvedené v [[petalinux-tools-documentation-2022](#)]. Pokud uživatel využívá starších verzí vývojových nástrojů, je doporučeno využít operační systém Linux. Pro tento 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 neplatí skutečnost, že když např. Vivado podporuje některou z novějších verzí Ubuntu, bude jí podporovat také PetaLinux Tools. Vždy je doporučeno využívat starší verze a kontrolovat vzájemnou kompatibilitu, aby se předešlo zbytečné ztrátě času.

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

## 10 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/vývoj 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 k 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 HLS. Díky tomuto systému složek bude pro každý projekt přesně stanovaná cesta k vytvořenému SDK (software development kit), který je potřeba pro vývoj akcelerované aplikace.

V této práci bude dodržována struktura naznačená v kódu ??.

```
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 10 - 1 Struktura složek, využívaná při tvorbě projektů k dosažení lepšího DX.

Tato struktura přináší možnosti rychlejšího pohyb v projektu pomocí vzdáleného přístupu SSH a emulátoru terminálu, který umožňuje provádět build aplikace i bez použití GUI Vitis HLS. Využíváním headless módu dochází k odstranění některých nedostatků SW, jako je např. zastavení build procesu z neznámých důvodů. 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.

## 11 Tvorba HW architektury Xilinx Vivado

Aby bylo možné vytvořit akcelerovanou aplikaci ve Vitis pomocí HLS C++, je třeba připravit platformu, resp. hardware, pro který bude daná aplikace vyvíjena. K tvorbě platformy je využit SW 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 vytvoření XSA souboru, který bude použit pro konfiguraci PetaLinux systému a 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 platforem 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.

### 11.1 Vivado Board Files

Aby bylo možné snadněji vytvořit potřebnou HW architekturu, firmy velmi často dodávají ke svému produktu *Board Files* soubory, obsahující různá přednastavení, konfigurace, informace a způsob připojení IP bloků k reálným součástím (constraints). [[github-vivado-board-files-for-digilent-fpga-boards](#)]

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 [[github-vivado-board-files-for-digilent-fpga-boards](#)]. Způsob instalace board files je popsáný v oficiální dokumentaci firmy Digilent, Inc. v [[digilent-installing-vivado-vitis-and-digilent-board-files](#)].

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

### 11.2 Tvorba HW designu pro Xilinx Kria KR260 vývojovou desku

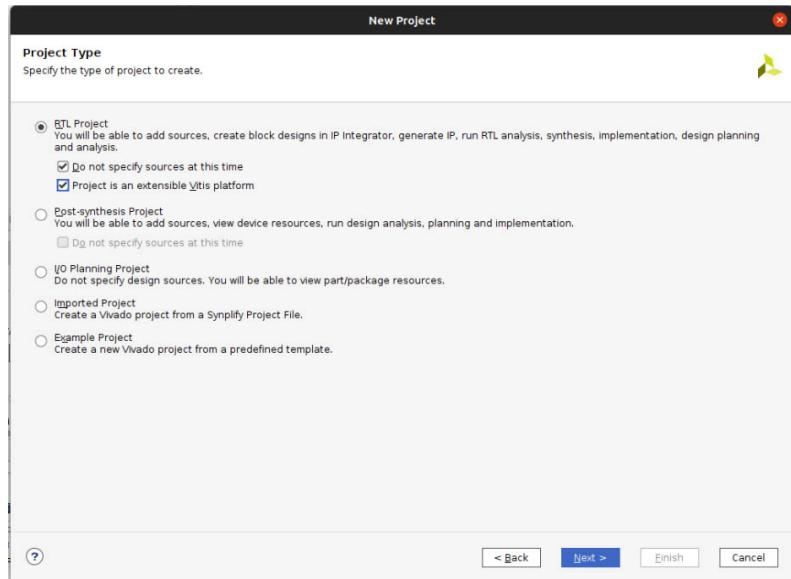
Při tvorbě designu pro vývojovou desku Xilinx Kria KR260, byly čerpány základní rámcové informace o postupu z [[hackster-getting-started-with-the-kria-kr260-in-petalinux](#)], [[hackster-add-peripheral-support-to-kria-kr260](#)] a [[hackster-getting-started-with-the-kria-kr260-in-vivado](#)]. Konkrétní postup se liší dle vytvářené aplikace a zkoumaných vlastností.

Protože první vývoj a zkoumání využitelnosti SoC bylo prováděno na desce Digilent Zybo Zynq-7000, je v příloze *Tvorba HW designu pro Digilent Zybo Zynq-7000 vývojovou desku* 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*).

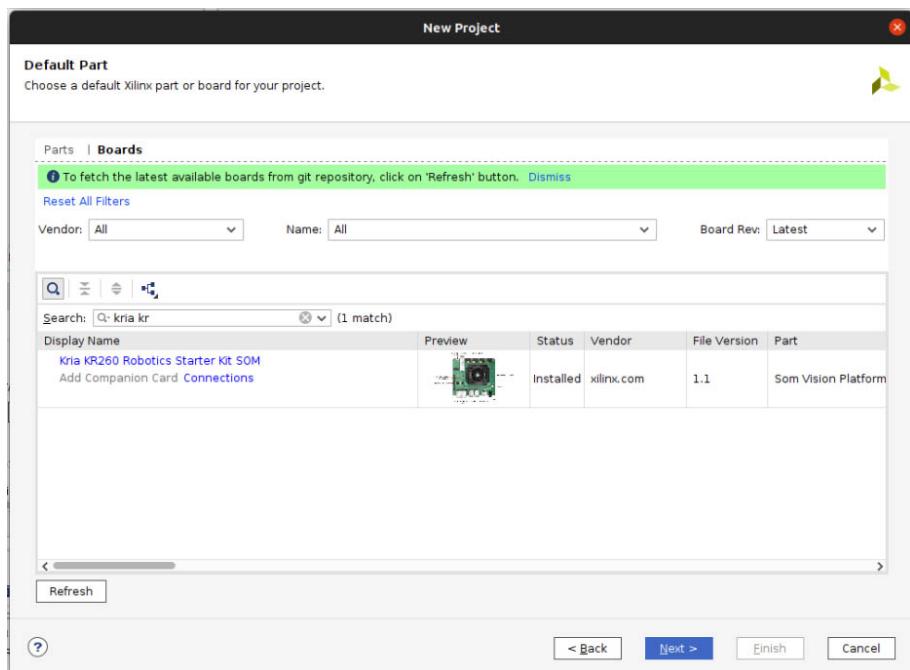
Postup tvorby projektu začíná pro většinu akcelerovaných aplikací stejným způsobem. Po otvěření programu Vivado stačí vytvořit nový project typu *RTL Project* a aktivovat nastavení *Project is an extensible Vitis platform*. Ukázka nabídky tvorby projektu je na obr. ??.

Dalším krokem při zakládání projektu je zvolení prvku, na který bude vyvíjený HW design určen. Je možné zvolit přímo komponentu, pro kterou je design určen nebo již přednastavené vývojové desky.



Obr. 11 - 1 Xilinx Vivado – volba typu projektu, použitelného dále jako platforma ve Vitis, pro Xilinx KR26.

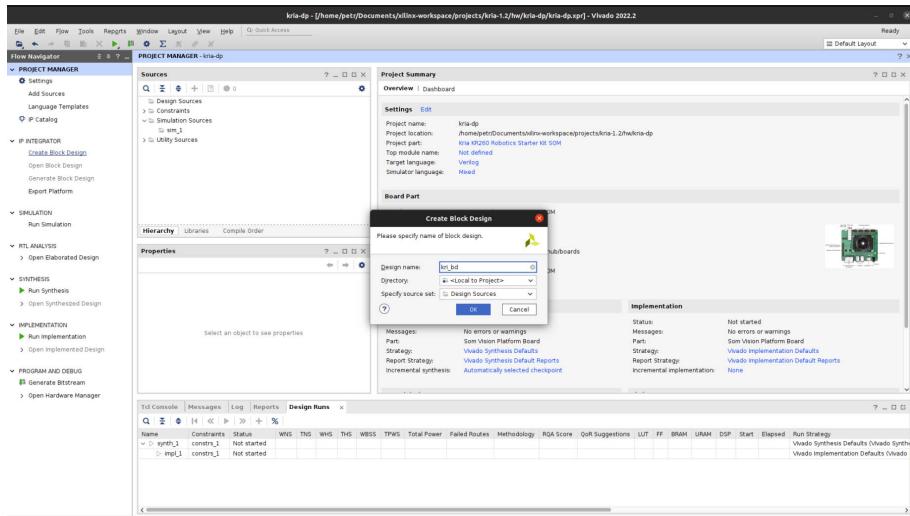
Pokud není vývojová deska v repozitáři od Xilinx, je možné ji vložit dle způsobu popsaného v části *Vivado Board Files*. Xilinx Kria KR260 je však již součástí daného repozitáře a je možné ji v repozitáři vyhledat a zvolit. Výběr desky z repozitáře je zobrazen na obr. ??.



Obr. 11 - 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 vytváření designu je třeba zvolit v menu odkaz *Create block design* a pojmenovat jej dle požadavků autora designu. Zvýrazněné menu a nabídka vytváření blokového je zobrazena na obr. ??.

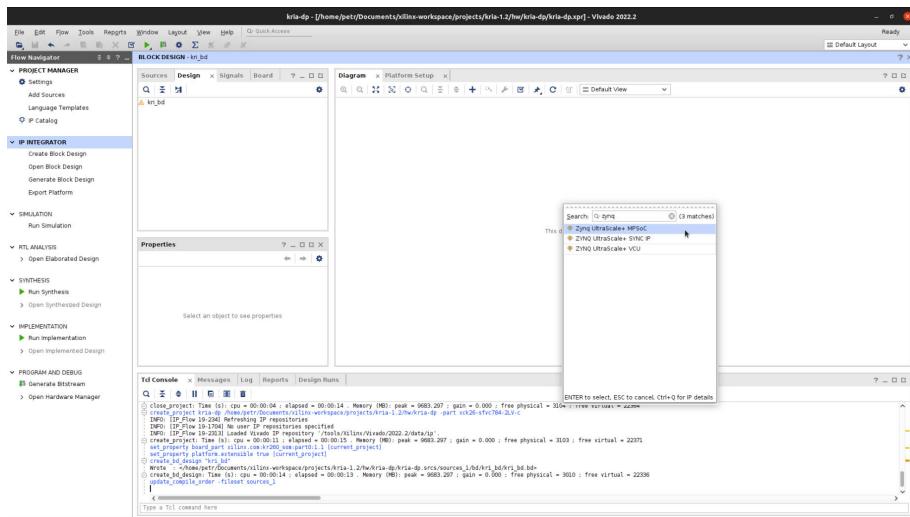
Až po krok vytváření block designu byl postup velmi podobný pro obě představené vývojové desky.



Obr. 11 - 3 Xilinx Vivado – nabídka vytváření block design.

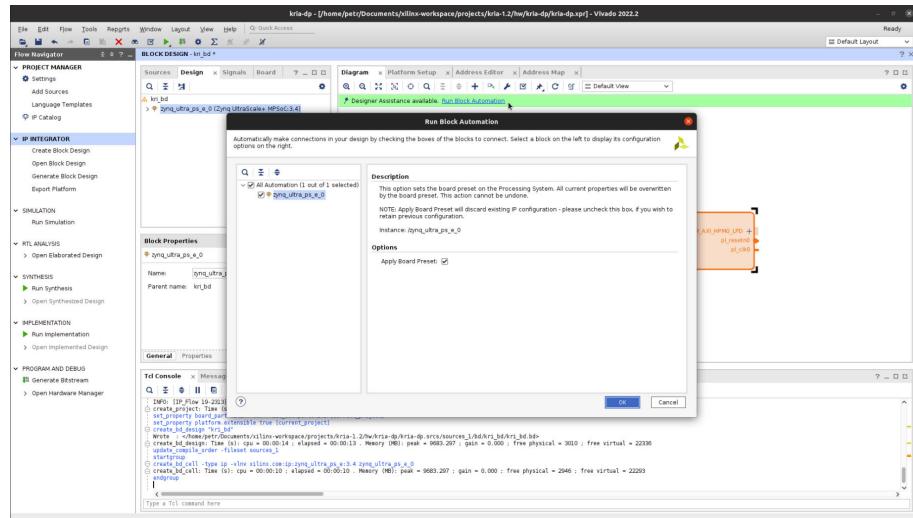
Nyní je již možné přistoupit k vlastní tvorbě blokového designu v kartě *Diagram*.

Bloky lze přidávat znakem + v aktivním okně nebo po kliknutí tlačítka myši do volného prostoru v téže okně a zvolení *Add IP* (IP – Intellectual Property). 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 vybraných bloků IP. Po vložení PS bloku je vhodné tuto automatizaci spustit pomocí aktivního odkazu, zobrazeného na kartě *Diagram* v obr. ??.



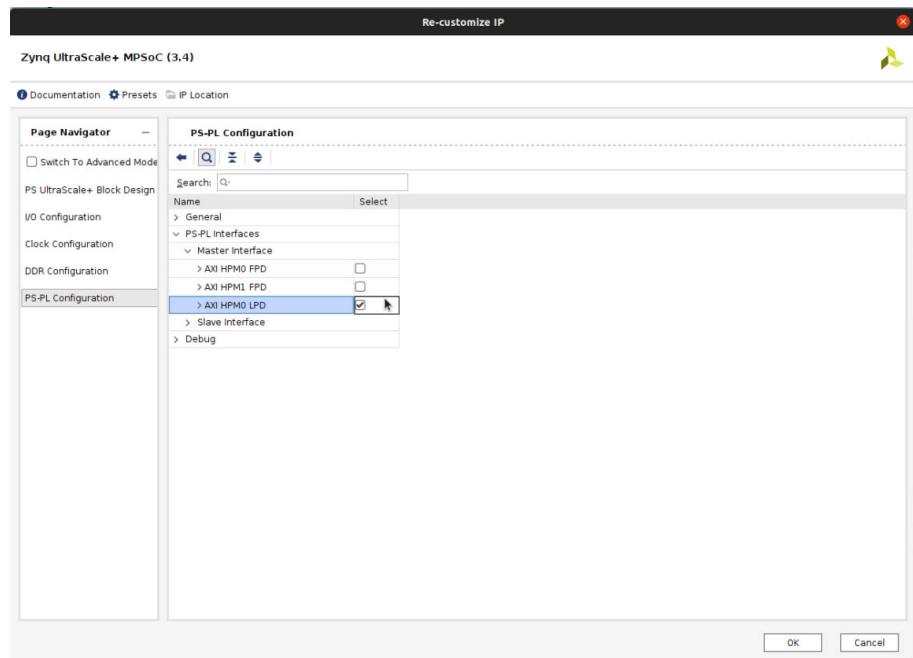
Obr. 11 - 4 Xilinx Vivado – vložení PS IP bloku.

Další krok je důležitý pro tvorbu akcelerované aplikace pomocí Vitis. Je třeba předkonfigurovat PS AXI Interface (rozhraní) takovým způsobem, aby AXI Interface 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 jak v [\[hackster-getting-started-with-the-kria-kr260-in-vivado\]](#) ale také v oficiálních příkladových [\[xilinx-github-vitis-tutorials-step-1-create-the-vivado-hardware-design-and-generate\]](#) (v sekci *Add Interrupt Support, krok 1*) pro tvorbu platformy pro vývojovou desku Xilinx Kria KV260, jež používá totožný modul Xilinx Kria K26. Obr. ?? obsahuje snímek obrazovky s požadovaným nastavením



Obr. 11 - 5 Xilinx Vivado – automatické propojení pro PS.

bloku Zynq UltraScale+ MPSoC a daných rozhraní.

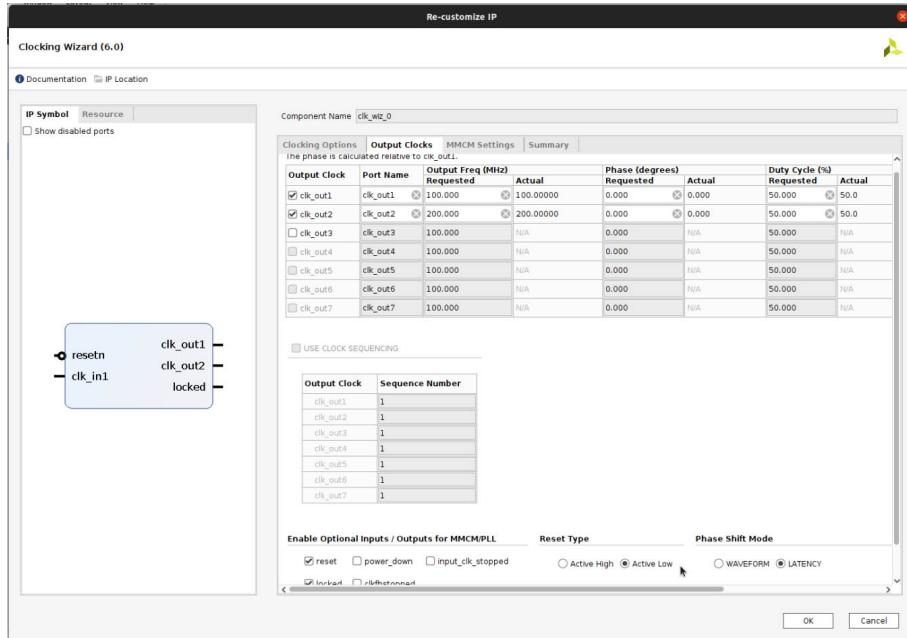


Obr. 11 - 6 Xilinx Vivado – zablokování FPD a odblokování LPD pro block design.

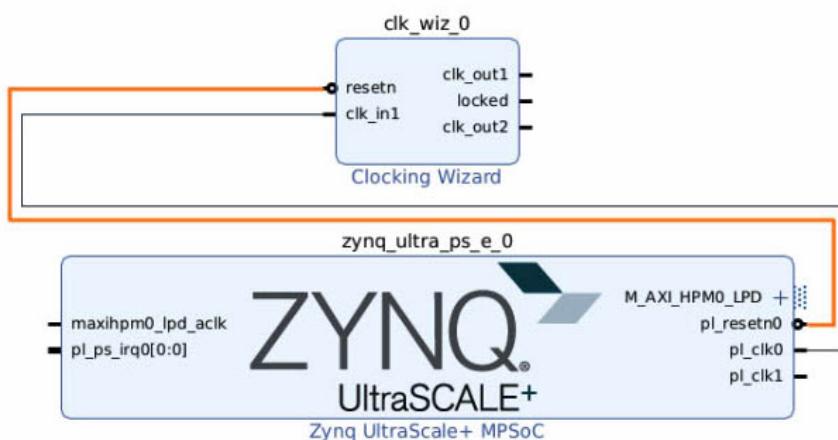
Protože dle [xilinx-github-vitis-tutorials-step-1-create-the-vivado-hardware-design-and-generate-xsa] je počet clock signálů `p1_c1k` z PS omezený, je vhodné pro vytvoření požadovaných taktovacích signálů využít blok **Clocking wizard**. V tomto bloku je možné vytvořit taktovací signály s požadovanými parametry. Pokud počet signálů nestačí, je možné přidat další IP.

Pro design ukázkové platformy v této práci je vhodné přidat 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. ??.

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. ??.

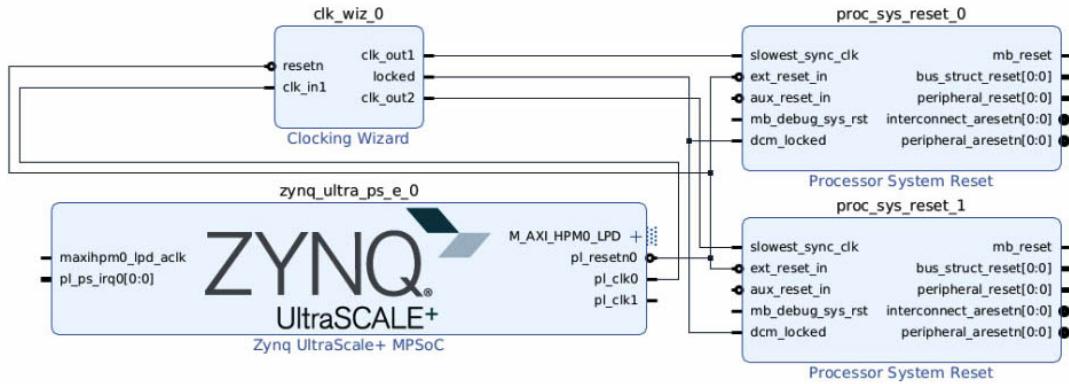


Obr. 11 - 7 Xilinx Vivado – nastavení bloku Clocking wizard.



Obr. 11 - 8 Xilinx Vivado – blokový design s PS a blokem Clocking wizard po provedení automatizace.

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



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

Vstup pro přerušení **pl\_ps\_irq** může obsahovat maximálně 16 interrupt signálů. Pro design v této aplikaci to je dostačující případ, ovšem pokud je vyžadováno, aby aplikace využívala více signálů přerušení, je třeba využít blok **AXI Interrupt Controller**. Doporučené nastavení tohoto IP je na obr. ???. Aby bylo možné připojit signál k **pl\_ps\_irq** je nutné přepnout nastavení *Processor Interrupt Type and Connection -> Interrupt Output Connection* na **Single** z výchozího **Bus**.

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

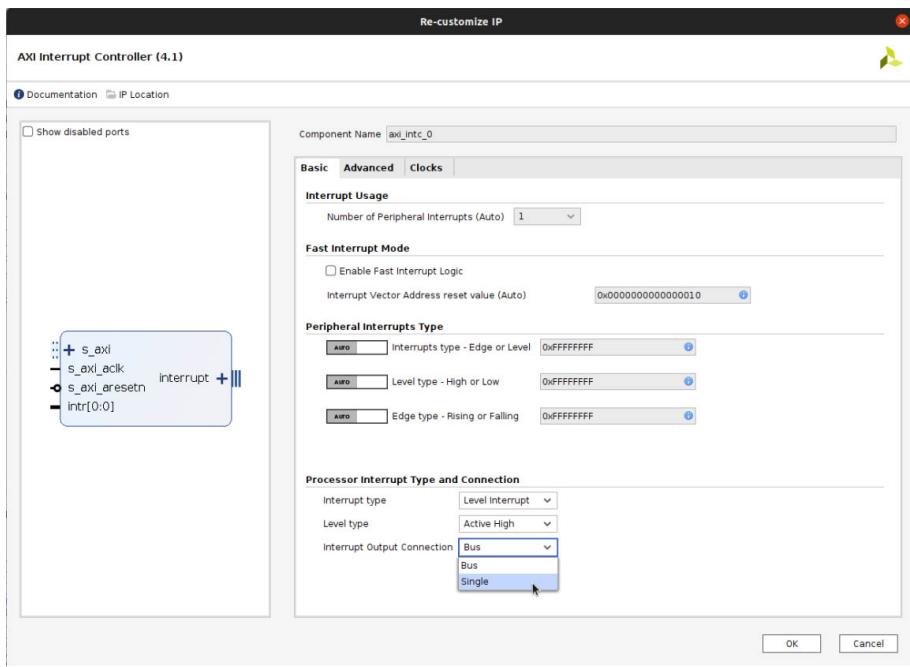
Po dokončení konfigurace **AXI Interrupt Controller** je opět možné aktivovat automatizaci propojení a v konfiguračním okně vybrat požadovanou frekvenci clock signálů. V této práci je využito 200 MHz. Výsledný design po automatizaci je zobrazen na obr. ??.

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

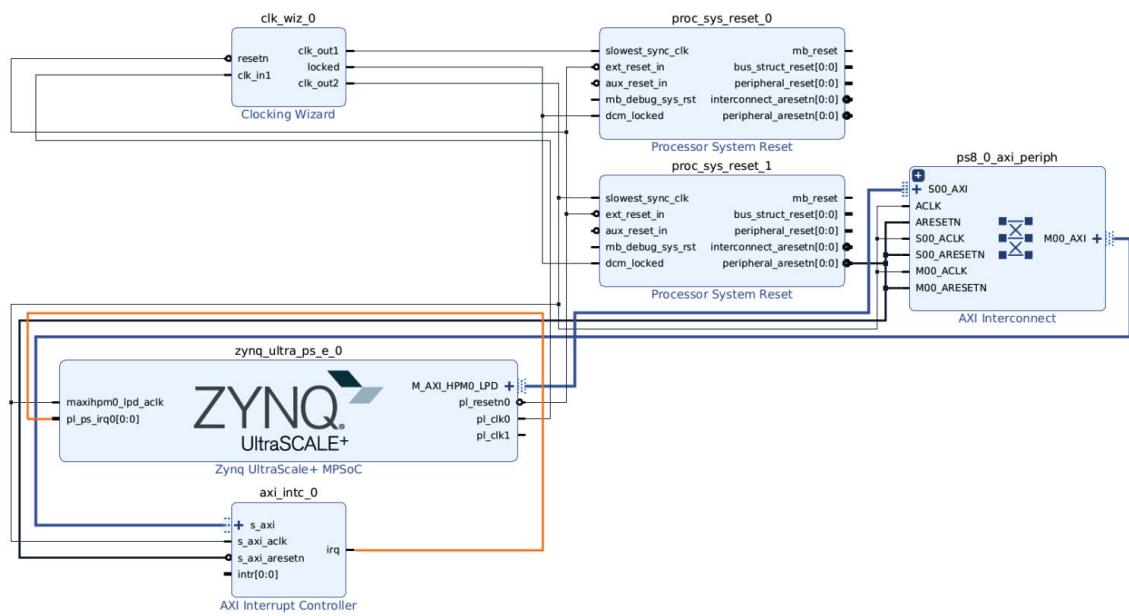
Ovšem při využití vývojové desky Xilinx Kria KR260 je vhodné aktivovat řízení chladícího ventilátoru pomocí PWM. Pro tuto možnost musí být aktivovaný výstup *Waveout* u TTC 0, jak je tomu naznámeno na obr. ???. Po aktivaci *Waveout* na I/O *EMIO* se objeví na bloku **Zynq UltraScale+ MPSoC** výstup s názvem *emio\_ttc0\_wave\_o[2:0]*. Označení *[2:0]* znamená, že signál je tříbitový, pro řízení ventilátoru pomocí PWM je ovšem využitelný pouze jeden bit. Pro odstranění dvou horních bitů je využito IP bloku **Slice**.

Konfigurace bloku **Slice** je zobrazena na obr. ???. *Din Width* určuje šířku vstupujícího signálu. V řešeném případě se jedná o trojbitový signál. *Din From* určuje označení bitu, od kterého bude odstraněno *Din Down To* bitů pro výstup z bloku **Slice**. Položka *Dout Width* je automaticky doplněna 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 myší kliknout na výstup bloku **Slice Dout[0:0]** a zvolit *Make External*. Tento pin je poté vhodné

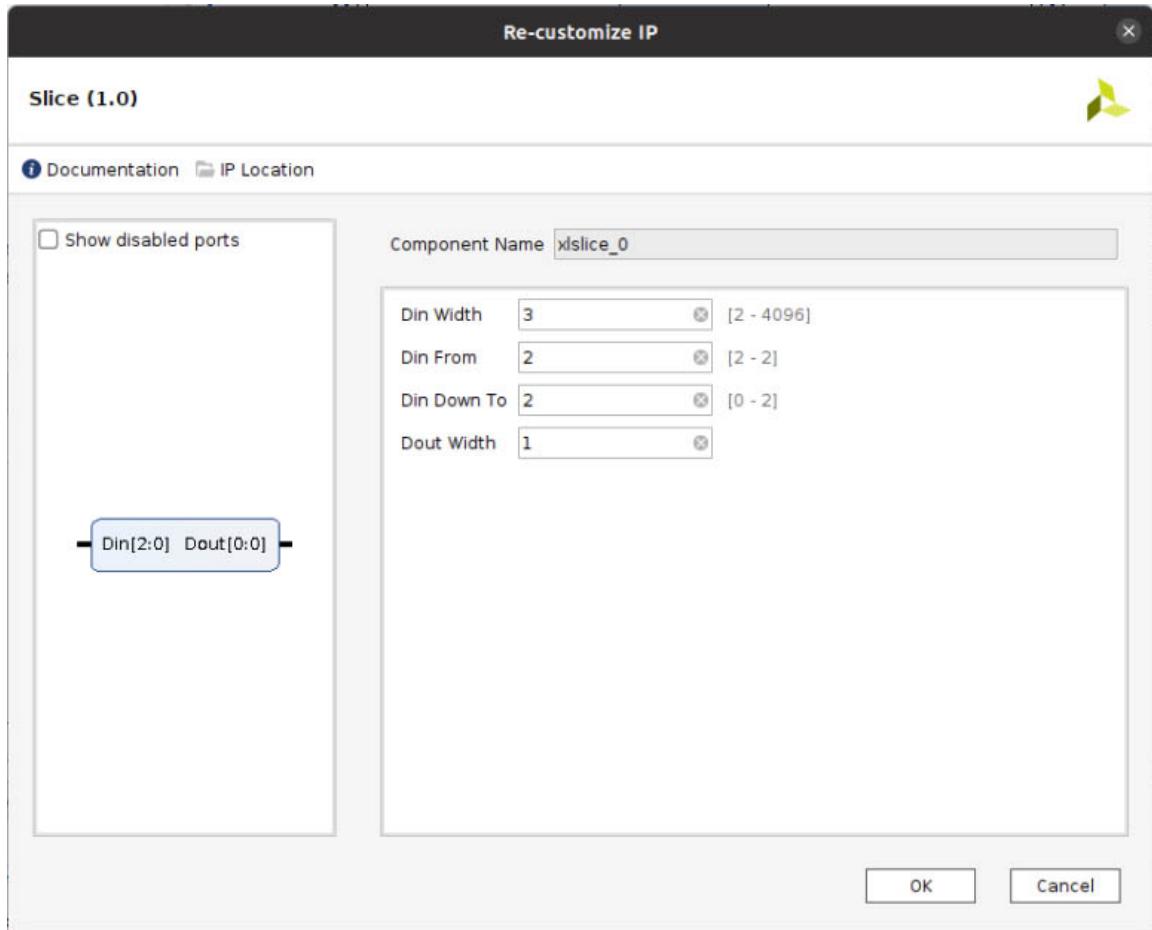


Obr. 11 - 10 Xilinx Vivado – nastavení bloku AXI Interrupt Controller.

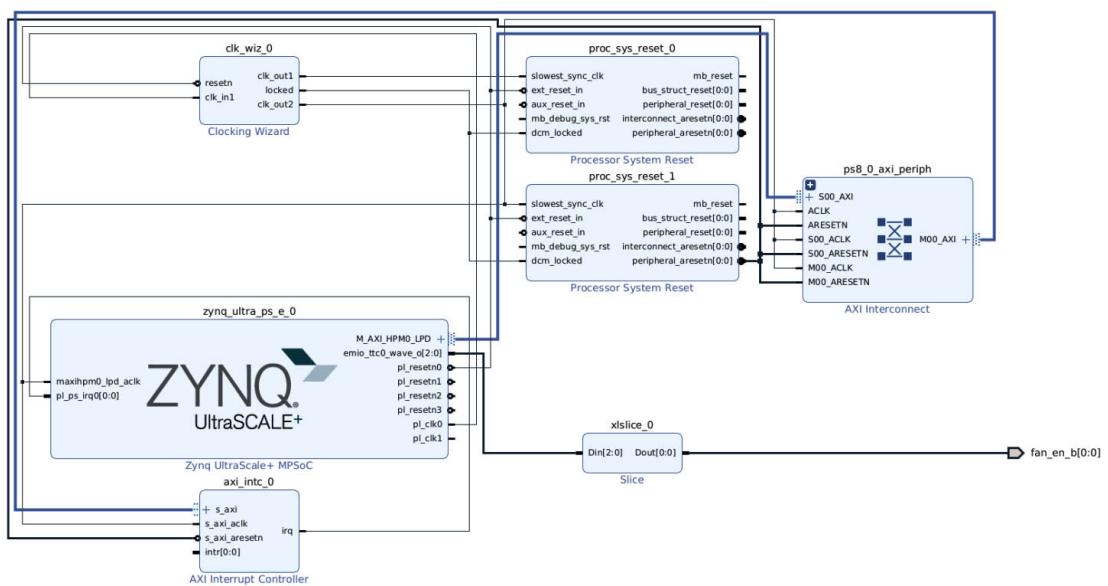


Obr. 11 - 11 Xilinx Vivado – block design po automatizaci a propojení bloku AXI Interrupt Controller.

pojmenovat `fan_en_b`. Minimální funkční design s výstupním pinem pro ventilátor je zobrazen na obr. ??.



Obr. 11 - 12 Xilinx Vivado – nastavení bloku Slice pro odstranění dvou horních bitů signálu.



Obr. 11 - 13 Xilinx Vivado – minimální funkční design s výstupním pinem pro řízení ventilátoru.

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

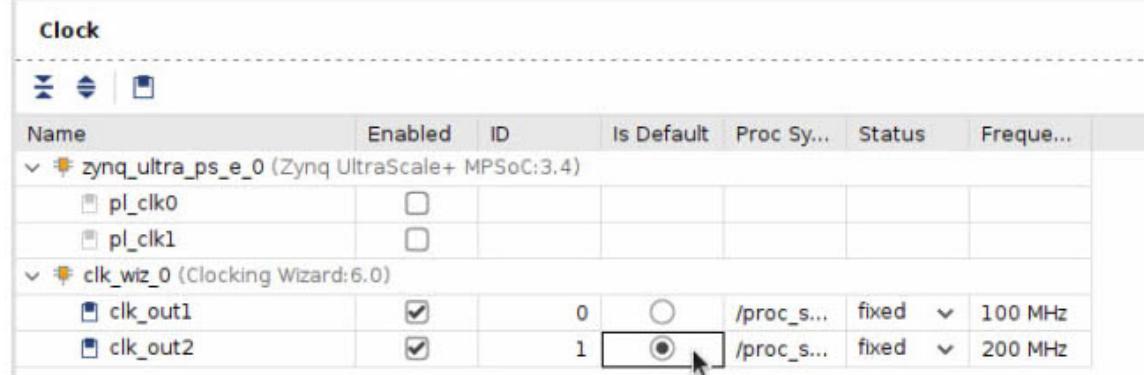
Tab. 11 - 1 Ukázka nastavených AXI portů v Xilinx Vivado platformě pro Xilinx Kria KR260.

| 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 clock signálů v záložce *Clock*. V této práci byl zvolen výchozí clock signál 200 MHz. Snímek záložky zachycující nastavení pro dva clock signály je na obr. ??.

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

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



Obr. 11 - 14 Xilinx Vivado – záložka nastavení clock signálů na platformě.

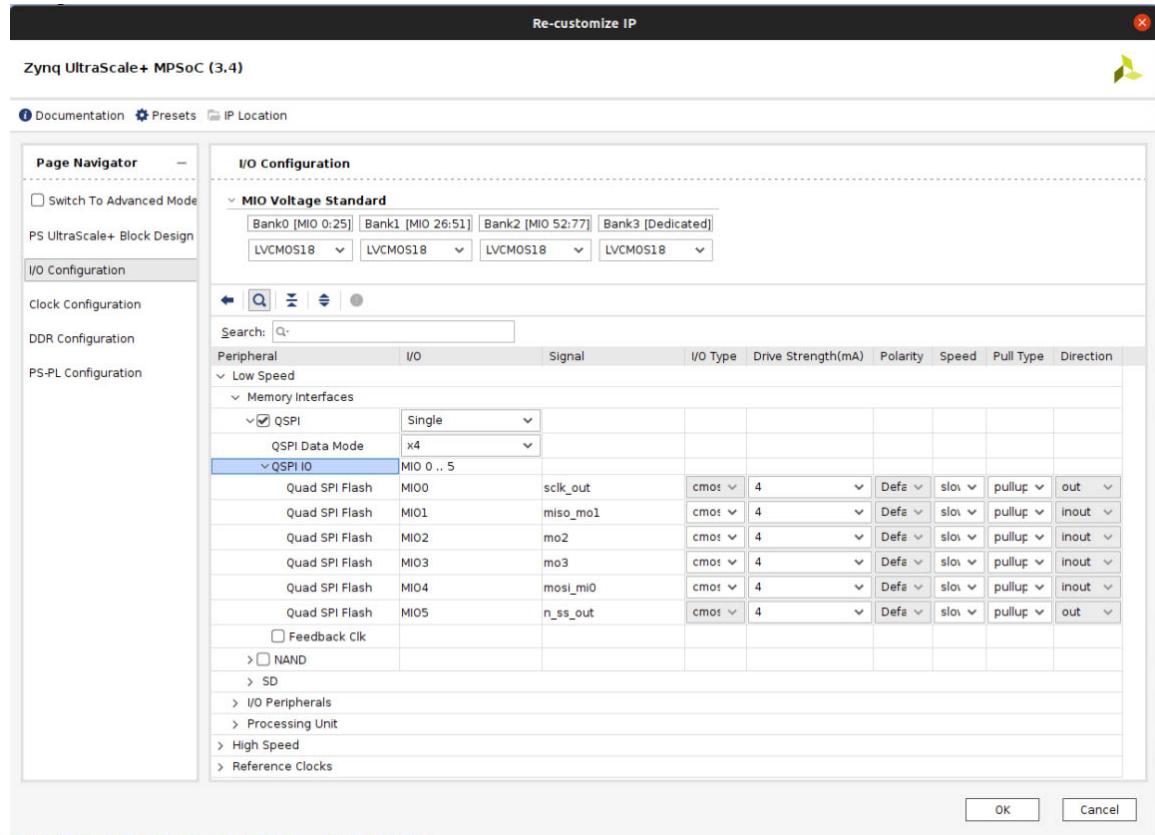
Autorka článku [[hackster-getting-started-with-the-kria-kr260-in-vivado](#)] zmiňuje, že pokud je zapnuta možnost *Incremental Synthesis*, můžou vznikat při postupu tvorby akcelerované aplikace problémy. Proto doporučuje pomocí nabídky *Settings -> Synthesis -> Incremental Synthesis* zvolit možnost *Disable incremental synthesis*.

### 11.2.1 Konfigurace PS pro využití implementovaných periferií

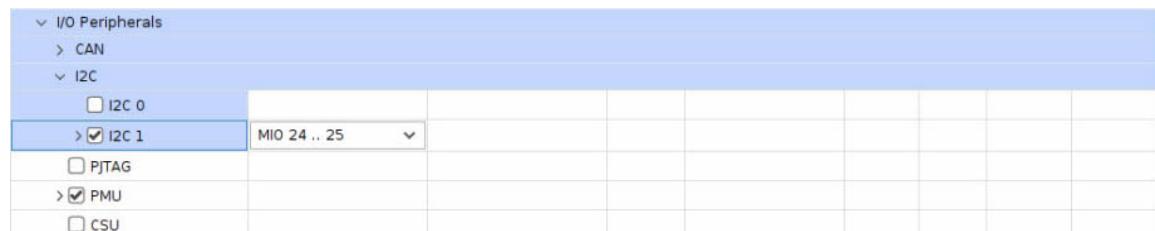
Aby bylo možné využívat periferie, jako je DisplayPort, Ethernet nebo USB konektory, je nutné nakonfigurovat PS. Konfigurační nabídka je otevřena po otevření IP bloku `Zynq UltraScale+ MPSoC` a vybrání `I/O Configuration`.

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

Na obr. ?? 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. 11 - 15 Xilinx Vivado – PS I/O Configuration QSPI zařízení.



Obr. 11 - 16 Xilinx Vivado – PS I/O Configuration I2C zařízení.

| Peripheral                        | I/O     | Signal | I/O Type | Drive Strength(mA) | Polarity | Speed | Pull Type | Direction |
|-----------------------------------|---------|--------|----------|--------------------|----------|-------|-----------|-----------|
| ✓ PMU                             |         |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPI EMIO |         |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPO EMIO |         |        |          |                    |          |       |           |           |
| > ✓ GPI 0                         | MIO 26  |        |          |                    |          |       |           |           |
| PMU GPI 0                         | MIO26   | gpi[0] | cmos     | 12                 | Defe     | fast  | pullup    | in        |
| <input type="checkbox"/> GPI 1    |         |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPI 2    |         |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPI 3    |         |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPI 4    |         |        |          |                    |          |       |           |           |
| > ✓ GPI 5                         | MIO 31  |        |          |                    |          |       |           |           |
| PMU GPI 5                         | MIO31   | gpi[5] | cmos     | 12                 | Defe     | fast  | pullup    | in        |
| <input type="checkbox"/> GPO 0    |         |        |          |                    |          |       |           |           |
| > ✓ GPO 1                         | MIO 33  |        |          |                    |          |       |           |           |
| PMU GPO 1                         | MIO33   | gpo[1] | cmos     | 4                  | Defe     | slow  | pullup    | out       |
| > ✓ GPO 2                         | MIO 34  |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPO 3    |         |        |          |                    |          |       |           |           |
| Initial State                     | GPO1[3] |        |          |                    |          |       |           |           |
| <input type="checkbox"/> GPO 4    |         |        |          |                    |          |       |           |           |

Obr. 11 - 17 Xilinx Vivado – PS I/O Configuration PMU (GPIO 4 a GPIO 5 není vybráno).

| v SPI                                     |             |             |      |   |      |      |        |       |
|---|-------------|-------------|------|---|------|------|--------|-------|
| > <input type="checkbox"/> SPI 0          |             |             |      |   |      |      |        |       |
| ✓ 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 | Defe | slow | pullup | inout |
| SPI 1                                     | MIO9        | n_ss_out[0] | cmos | 4 | Defe | slow | pullup | inout |
| SPI 1                                     | MIO10       | miso        | cmos | 4 | Defe | slow | pullup | inout |
| SPI 1                                     | MIO11       | mosi        | cmos | 4 | Defe | slow | pullup | inout |

Obr. 11 - 18 Xilinx Vivado – PS I/O Configuration SPI zařízení.

|                                |              |     |      |    |      |      |        |     |
|--------------------------------|--------------|-----|------|----|------|------|--------|-----|
| ✓ UART 1                       | MIO 36 .. 37 |     |      |    |      |      |        |     |
| <input type="checkbox"/> MODEM |              |     |      |    |      |      |        |     |
| UART 1                         | MIO36        | txd | cmos | 4  | Defe | slow | pullup | out |
| UART 1                         | MIO37        | rxd | cmos | 12 | Defe | fast | pullup | in  |

Obr. 11 - 19 Xilinx Vivado – PS I/O Configuration UART zařízení.

| v GPIO                             |              |  |  |  |  |  |  |  |
|------------------------------------|--------------|--|--|--|--|--|--|--|
| <input type="checkbox"/> GPIO EMIO |              |  |  |  |  |  |  |  |
| > ✓ GPIO0 MIO                      | MIO 0 .. 25  |  |  |  |  |  |  |  |
| > ✓ GPIO1 MIO                      | MIO 26 .. 51 |  |  |  |  |  |  |  |
| <input type="checkbox"/> GPIO2 MIO |              |  |  |  |  |  |  |  |

Obr. 11 - 20 Xilinx Vivado – PS I/O Configuration GPIO zařízení.

| v Processing Unit                  |  |  |  |  |  |  |  |  |
|------------------------------------|--|--|--|--|--|--|--|--|
| v SWDT                             |  |  |  |  |  |  |  |  |
| > ✓ SWDT 0                         |  |  |  |  |  |  |  |  |
| <input type="checkbox"/> Clock in  |  |  |  |  |  |  |  |  |
| <input type="checkbox"/> Reset out |  |  |  |  |  |  |  |  |
| > ✓ SWDT 1                         |  |  |  |  |  |  |  |  |
| <input type="checkbox"/> Clock in  |  |  |  |  |  |  |  |  |
| <input type="checkbox"/> Reset out |  |  |  |  |  |  |  |  |

Obr. 11 - 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. 11 - 22 Xilinx Vivado – PS I/O Configuration Triple Timer Counter (TTC), TTC 0 výstup Waveout je využit pro PIN řídící chladící 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. 11 - 23 Xilinx Vivado – PS I/O Configuration USB.

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

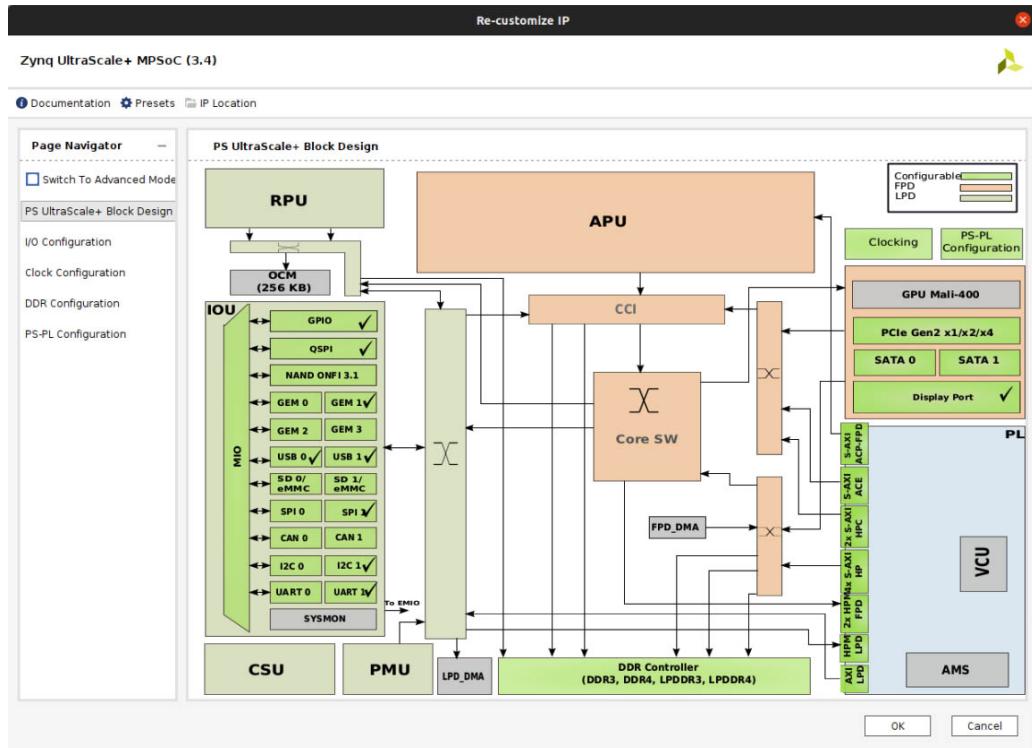
Obr. 11 - 24 Xilinx Vivado – PS I/O Configuration DisplayPort.

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.

Toto nastavení *Number of Fabric Resets* udává, kolik signálů je možné použít pro reset PL IP bloků.

[[xilinx-ultra-scale-plus-mpsoc-processing-syste-product-guide](#)], [[xilinx-wiki-atlassian-zynq-ultra-scale-plus-mpsoc-re](#)]

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



Obr. 11 - 25 Xilinx Vivado – PS UltraScale+ Block Design v PS IP.

### 11.2.2 Design Constraints

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

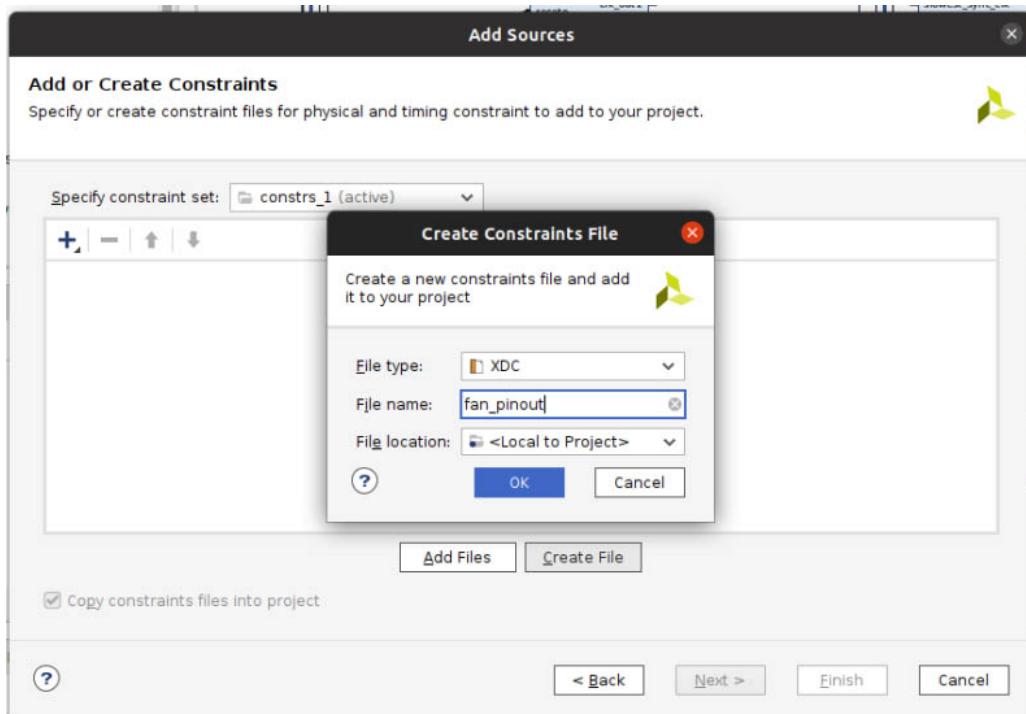
Jejich tvorba může probíhat i mimo prostředí Vivado, jsou to textové soubory, jež lze upravit v libovolném textovém editoru. Tato skutečnost lze využít při časté tvorbě designů v nových projektech pro zrychlení konfigurace. Soubory je poté možné jednoduše importovat.

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

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 ??.

Kód ?? byl získán z [[hackster-add-peripheral-support-to-kria-kr260-vivado](#)]. Autorem však bylo ověřeno, že se skutečně jedná o fyzický pin A12.

```
1 set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```



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

```

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 II - 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 také 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 [[xilinx-support-forum-petrzakopal-schematics](#)]

Postup nalezení reálného pinu je následující:

1. Nalézt ve schématu [[kria-kr260-starter-kit-cc-schematics](#)] Kria KR260 CC požadované označení fyzického pinu (např. pro ventilátor HDA20).
2. Nalézt ve schématu [[kria-kr260-starter-kit-cc-schematics](#)], ke kterému pinu na PL connector (**SOM240\_1 CONNECTOR**) je nalezený PIN připojen. (v případě ventilátoru C24).
3. V XDC [[kria-k26-som-xdc](#)] souboru pro Xilinx Kria K26 vyhledat daný pin z connector připojení, zkontovalovat, zda je připojen ke správnému connectoru ze schématu a získat požadovanou hodnotu 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 provedení akce je možné zvolit opět v nabídce pro block design akci *Create HDL Wrapper*. V nabídce akce je výhodné využít možnosti *Let Vivado manage wrapper and auto update*, kdy Vivado bude aktualizovat HDL wrapper podle změn provedených v designu.

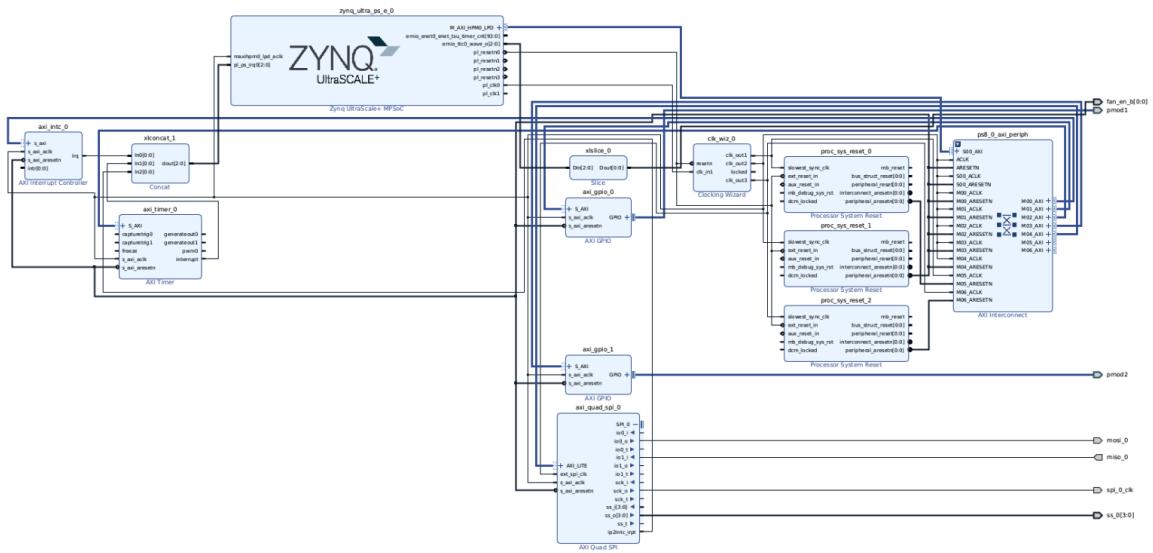
Autorovi se ovšem při změně block designu vyplatilo HDL Wrapper kompletně smazat a opětovaně provést kroky *Generate Output Products* a *Create HDL Wrapper*.

Nyní je již možné přistoupit k syntéze, implementaci a generování bit stream. 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.

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*, pokud je požadovaná také HW emulace. Dle [\[hackster-add-peripheral-support-to-kria-kr260-vivado\]](#) podpora HW emulace pro K26 prozatím není dostupná. Posledním krokem je vybrání v nastavení *Platform State* volby *Pre-synthesis* a *Include bitstream*. Po nastavení 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.

### 11.2.3 HW block design vyvýjené aplikace



Obr. 11 - 27 Xilinx Vivado – HW block design vyvíjené aplikace.

## 12 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 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 osabující prvky představené v části *HW block design vyvíjené aplikace*.

Postup tvorby *PetaLinux* systému čerpá informace z [[hackster-getting-started-with-the-kria-kr260-in-petalinux](#)], [[xilinx-github-vitis-tutorials-step-2-create-the-software-components](#)] 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 ???. Představený postup předpokládá práci s definovanou strukturou vyvíjené aplikace/projektu definované v části *Struktura složek* v kódu ???.

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

*Kód 12 - 1 Aktivace prostředí PetaLinux verze 2022.2.*

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

```
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 12 - 2 Tvorba PetaLinux projektu ze základního BSP souboru.*

### 12.1 Hardware konfigurace PetaLinux

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

První konfigurace spočívá v implementaci HW platformy do projektu pomocí příkazu ???. 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 12 - 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 ???.

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

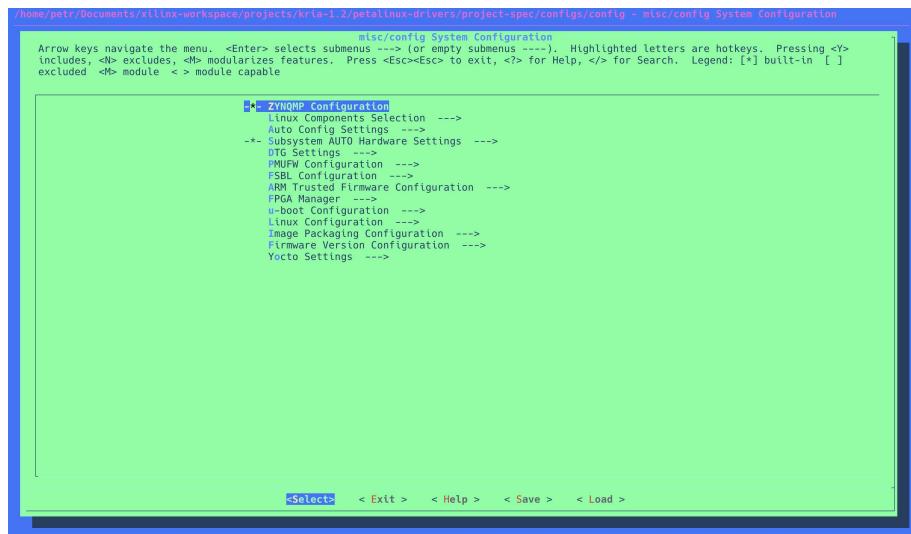
```

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

*Kód 12 - 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. ??.



*Obr. 12 - 1 Xilinx Vivado – snímek obrazovky konfigurace PetaLinux pomocí „petalinux-config“ příkazu.*

Po prvotní konfiguraci bylo zjištěno, že při požadavku na aplikaci RT patch pro PetaLinux je vhodné provést prvotní build systému a poté pokračovat v následných konfiguracích. Build proces je spuštěn příkazem ??.

Proces aplikace RT Patch je představen v části *Postup aplikace PREEMPT\_RT patch*.

```
1 petalinux-build
```

*Kód 12 - 5 PetaLinux build příkaz pro vytvoření systému.*

## 12.2 Konfigurace kernelu PetaLinux

Po aplikaci patche je vhodné pokračovat v nastavování kernelu PetaLinux systému pomocí příkazu ?? dle požadavků aplikace. Ve vytvářené aplikaci je využíván `Userspace IO Driver` a proto je nutné aktivovat konfiguraci kernelu, která zajistí automatické načtení driveru (ovladače) do systému při jeho startu. Pokud by se tak nestalo, bylo by nutné vkládat moduly, resp. drivery manuálně pomocí příkazů `modprobe` či `insmod`. Při konfiguraci kernelu je možné aktivovat i další prvky, které nejsou v této práci využívány. Aktivovaná konfigurace pro `Userspace IO Driver` je zobrazena v kódu ??.

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

*Kód 12 - 6 PetaLinux příkaz pro konfiguraci kernelu 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 12 - 7 PetaLinux konfigurace pro User Space IO Drivers.

### 12.2.1 Konfigurace Device Tree

Aby bylo možné využívat zmiňovaný **Userspace IO Driver** v podobě **generic-uio** driver pro řešení přerušení (interrupts) od prvků AXI Quad SPI [[axi-quad-spi-ip-product-guide](#)], AXI Timer [[axi-timer-v-2-0-ip-product-guide](#)] a dalších, je nezbytné provést 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 jsou vydávány specifikace, které tuto datovou strukturu popisují. Tyto specifikace je možné nalézt na oficiálních stránkách této iniciativy. [[devicetree-org-specification](#)]

Existuje i záznam velmi přínosné prezentace z *Embedded Linux Conference Europe* ohledně problematiky Device Tree od *Thomas Petazzoni* s názvem *Device Tree for Dummies!* [[youtube-devicetree-for-dummies](#)].

Při tvorbě akcelerované aplikace v této práci byl využit **devicetree** soubor automaticky vytvořený při konfiguraci kernelu a prvním build procesu *PetaLinux* systému. Ovšem aby bylo možné použít zmiňované bloky, je třeba do souboru **system-user.dtsi**, který slouží k překonfigurování automaticky vytvořeného souboru **system-conf.dtsi**, protože je při **parse** procesu čten jako poslední.

Obsah **system-user.dtsi** souboru je zobrazen v ???. 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  
8     timer@0080020000 {  
9         compatible = "axi_timer_0, generic-uio, ui_pdrv";  
10        status = "okay";  
11    };  
12  
13    axi_quad_spi@80040000 {  
14        compatible = "axi_quad_spi_0, generic-uio, ui_pdrv";  
15        status = "okay";  
16    };  
17};
```

Kód 12 - 8 Obsah souboru **system-user.dtsi**, překonfigurovávající soubor **system-conf.dtsi**.

Tento rekonfigurační **devicetree** byl získán reengineeringem souboru **pl.dtsi**. Soubor **pl.dtsi** je získán za pomoci exportovaného *XPR* souboru pomocí **XSCT** (Xilinx Software Command-Line Tool). Postup byl získán ze zdrojů [[xilinx-github-vitis-tutorials-step-2-create-the-software-components](#)] a [[hackster-getting-started-with-the-kria-kr260-in-petalinux](#)].

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

```

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 12 - 9 Postup tvorby pl.dtsi souboru pomocí XSCT, popisující devicetree v runtime.*

Uvedeným způsobem je získán textový soubor `pl.dtsi`, který je možné opět konfigurovat pomocí textového editoru. Soubor má podobnou strukturu jako běžný `devicetree`, ovšem definuje tzv. fragmenty pro `device tree overlay`. Protože byla aktivována v kroku HW konfigurace *PetaLinux* možnost `FPGA Manager -> Fpga Manager <*>`, je v jádru systému *PetaLinux* zabudovaná jen základní část `devicetree`. Zbytková část je poté načtena do systému při běhu systému (at runtime) při 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 nutné u použitého bloku AXI Timer změnit *property value compatible = "generic-uio"*; a pro AXI Quad SPI na téže hodnotu.

Následně je nutné textový soubor `pl.dtsi` převést do binární podoby, vhodné pro systém. Ke kompliaci je používán nástroj `dtc` (Device Tree Compiler), dostupný pro většinu distribucí Linux. Příkaz, který popisuje kompliaci souboru `pl.dtsi` (Device Tree Source Include) na `pl.dtbo` (Device Tree Blob Object) je zobrazen v kódu ??.

```

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 12 - 10 Kompilace textového souboru device tree overlay pl.dtsi na binární pl.dtbo*

Pokud byla kompliacie 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`.

## 12.3 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 ??.

```
1 petalinux-config -c rootfs
```

Kód 12 - 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 ??.

Pokud vývoj aplikace nevyžaduje nejnovější instalační balíky a jejich aktualizace pomocí 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 mj. zajišťující komunikaci mezi PS aplikací a akcelerovanými kernaly v PL,

`zocl` – ovladač pro ZynQ OpenCL akcelerátory,

`opencl-headers` a `opencl-c1hpp` – 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).

Aktivované nastavení automatického přihlašování není doporučeno používat v zařízení v provozu z bezpečnostních důvodů. Při debuggingu a vyvíjení aplikace ovšem přináší zjednodušení přihlašování. 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 12 - 12 Doporučené nastavení rootfs pro akcelerovanou aplikaci.*

## 12.4 Závěrečný build PetaLinux, generování SDK a tvoření WIC obrazu systému

Po provedení požadované `rootfs` konfigurace je možné přistoupit opět k build procesu pomocí příkazu `??`. Po první části build procesu je možné aktivovat proces generování *SDK* (Software Development Kit), který je nutný k tomu, aby bylo možné akcelerovanou aplikaci ve Vitis vyvíjet a aktivovat její build proces. Generování *SDK* je provedeno pomocí příkazu `??`.

Tento *SDK* je pro jeho využití ve Vitis IDE nutné instalovat. Instalace je doporučena do složky `linux-files`, udaná strukturou v části *Struktura složek* pomocí příkazu `??`. Po vygenerování *SDK* je možné přistoupit ke kroku vytvoření a „zabalení“ (package) určitých komponent, pro vytvoření obrazu systému *PetaLinux*. K tomuto jsou využity příkazy `??` a `??`.

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

*Kód 12 - 13 Příkaz pro aktivování build procesu SDK*

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

*Kód 12 - 14 Příkaz pro instalaci SDK*

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

*Kód 12 - 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 12 - 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 snadno vložené. Boot komponenty jsou umístěny v cestě `<petalinux-project>/images/linux` a je vhodné je zkopírovat do složky ve vytvořené struktuře dle příkazu `??`.

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

Kód 12 - 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 následně vložena do 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 vyvíjenou aplikaci. Na závěr části byl nastíněn postup flash SD karty, kterou je poté možné vložit do vývojové desky a započít boot proces systému.

## 13 Tvorba akcelerované aplikace ve Vitis IDE

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

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

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

Aby bylo možné Vitis IDE spustit v operačním systému Linux, je nutné opět aktivovat prostředí pomocí příkazu **??**. 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*.

### 13.1 Tvorba platformy pro akcelerovanou aplikaci

Před tvorbou samotné akcelerované aplikace je nejprve nutné vytvořit tzv. *Platform project*, jež využije vytvořený soubor *XSA* z Vivado a soubor *Image* z cesty `<petalinux-project>/linux/image/`*Image*. V nabídce výběru systému při tvorbě platformy je nutné vybrat *linux* a zrušit možnost *Generate boot components*. Konfigurace je zobrazená na obr. **??**.

Po vytvoření platformy je nutné 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 **??**.

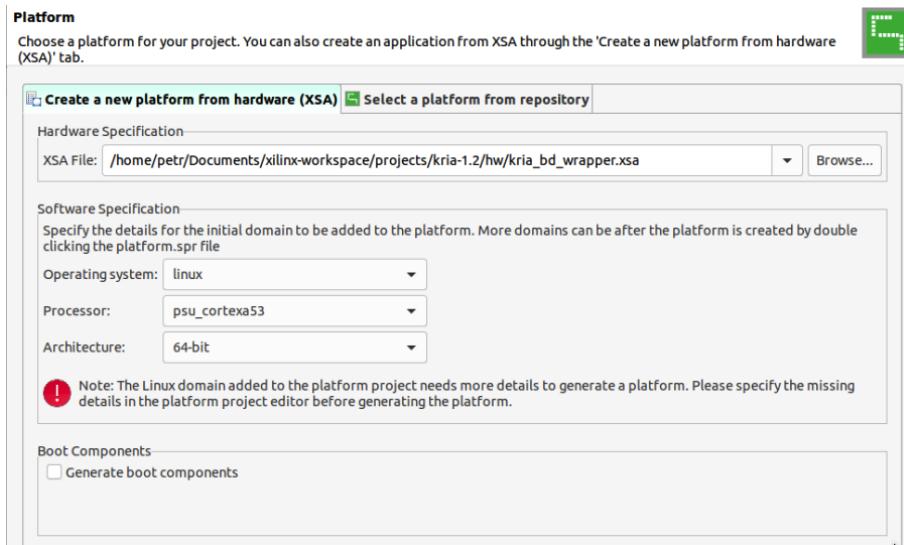
```
source /tools/Xilinx/Vitis/2022.2/settings64.sh
```

Kód 13 - 1 Aktivace prostředí pro Vitis verze 2022.2.

Tab. 13 - 1 Nastavení cest souborů pro platformu ve Vitis IDE souboru *platform.spr*.

| Jméno                     | Cesta  | Pozmánka                        |
|---------------------------|--|---------------------------------|
| 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 | <code>&lt;project-name&gt;/linux-files/pfm/boot</code>                                   | -                               |
| Linux Rootfs              | <code>&lt;project-name&gt;/&lt;petalinux-project&gt;/images/linux/rootfs.ext4</code>     | generované pomocí <i>SDK</i>    |
| Bootmode                  | SD   | generované automaticky          |
| FAT32 Partition Directory | <code>&lt;project-name&gt;/linux-files/pfm/sd_card</code>                                | složka zůstane prázdná          |
| Sysroot Directory         | <code>&lt;project-name&gt;/linux-files/sysroots/cortexa72-cortexa53-xilinx-linux/</code> | generované pomocí <i>SDK</i>    |
| QEMU Data                 | -  | generované automaticky          |
| QEMU Arguments            | -  | generované automaticky          |
| PMU QEMU Arguments        | -  | generované automaticky          |

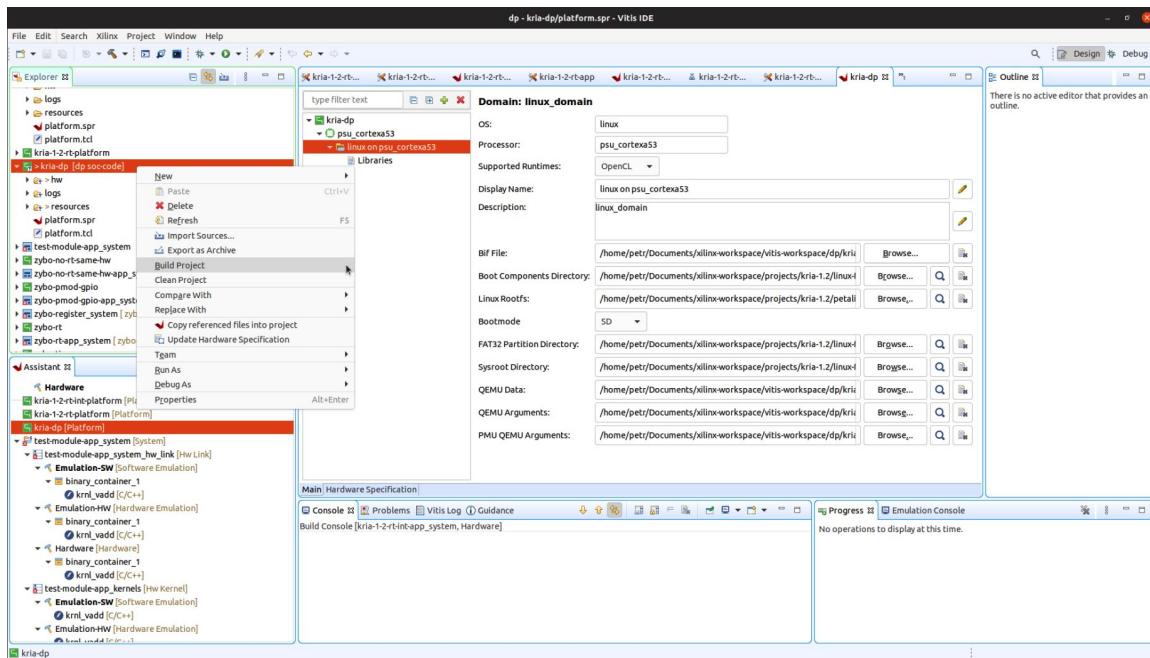
Jak již bylo řešeno, většinu souborů ve Vitis je možné otevřít pomocí textového editoru. Soubor *platform.spr* není výjimkou. Pokud dochází k tvorbě mnohočetných projektů se stejným nastavením platformy, je možné vytvořit skripty, které automaticky vytvoří konfigurovaný *platform.spr* soubor



Obr. 13 - 1 Xilinx Vitis IDE – tvorba platformy pro akcelerovanou aplikaci.

s danou strukturou a nastavením.

Po základní konfiguraci platformy je možné spustit komplikaci platformy pomocí pravého kliknutí na název platformy v okně *Explorer* a zvolení možnosti *Build Project*. Ukázka tohoto kroku je na obr. ??.



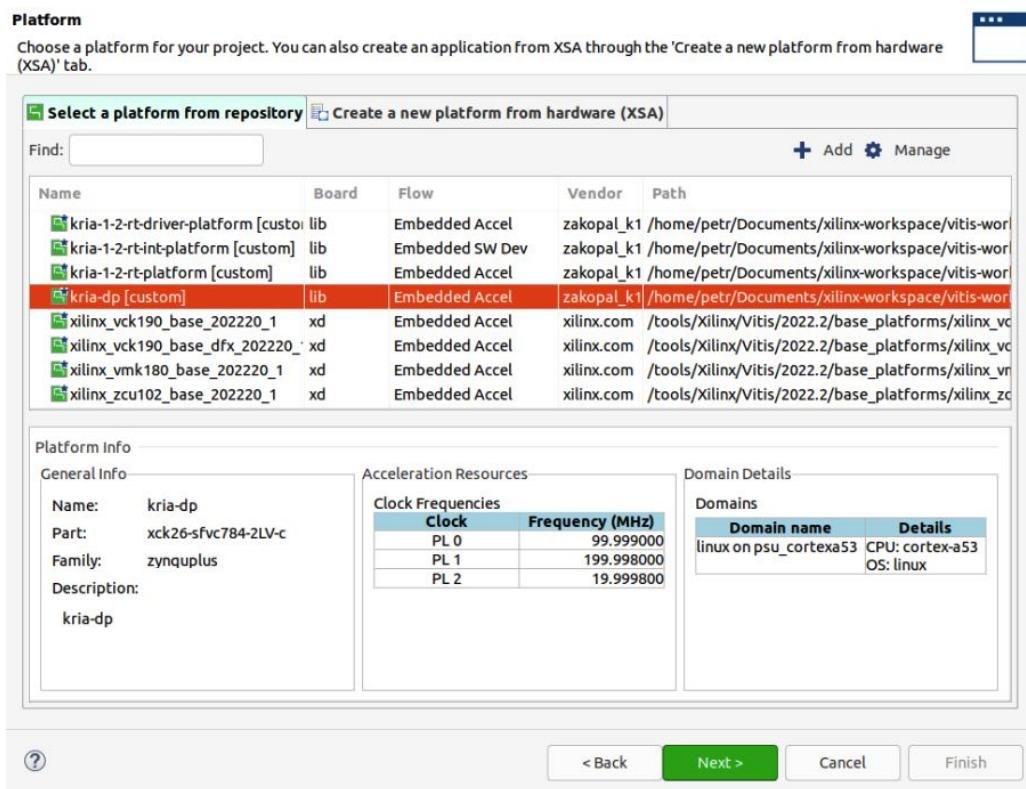
Obr. 13 - 2 Xilinx Vitis IDE – build proces platfromy.

## 13.2 Tvorba application projektu

Po dokončení build procesu platformy je již možné přejít k tvorbě *Application Project*. *Application Project* je možné vytvářen pouze na předem vytvořenou platformu, která prošla úspěšným build procesem. Ukázka výběru platformy s povolenou HW akcelerací, na kterou může být vytvářen *Application Project* je na obr. ??.

Při inicializaci projektu v kroku *Domain* je nutné zvolit v části *Application settings* cestu k položce

*Kernel Image*, ta je pro vyvýjenou aplikaci umístěna v cestě <project-name>/<petalinux-project>/image



Obr. 13 - 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 soubory projektu předkonfigurovány a je možné vhodně měnit, aby splňovali potřebné parametry.

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 artefakty pro PL do jednoho binárního souboru `xclbin`. Makefile soubory je možné vytvořit manuálně, nebo pomocí Vitis IDE v okně *Assistant* poklikem levým tlačítkem myši na název potřebného typu procesu build a volbou *Create Makefiles*. Toto je potřeba provést nejméně 4x. 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 téže 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 akci 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 přidat nové soubory pro generování a linkování objektů pro kernel.

Doporučený postup build pro *Hardware* build je následující:

1. build host aplikace pomocí `makefile` v cestě `<vitis-workspace>/<application-name>/Hardware/makefile` (nejrychlejší proces),
2. build kernelu pomocí `makefile` v cestě `<vitis-workspace>/<application-name>_kernels/Hardware/makefile` (středně pomalý proces),
3. link objektů `xo` z kroku č. 2 do objektu `xclbin` pro PL pomocí `makefile` souboru v cestě `<vitis-workspace>/<application-name>_system_hw_link/Hardware/makefile` (nejpomalejší proces, obsahuje syntézu, implementaci, generování bitstreamu), tento proces u vytárené aplikace v této práci 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 při otevření konfiguračního souboru v okně *Explorer* v cestě `<application-name>_system/<application-name>_kernels/<application-name>_kernels.prj` a výběru **Hardware** v nastavení pro *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, je nutné mj. pro jejich zkompilování je umístit i do tohoto souboru pomocí GUI nabídky. Ovvšem v některých případech nedochází ke správné indexaci souborů ze strany Vitis IDE a je nutné upravit soubor v textovém editoru.

## 14 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 předchozího kroku *Konfigurace Device Tree* byl získán soubor `p1.dtbo`, který byl přemístěn do složky `transfer`, definované v *Struktura složek*. Do téže složky je vhodné překopírovat soubory:

- `<application-name>/Hardware/<application-name>` (host program),
  - `<application-name>_system_hw_link/Hardware/<binary-container>.xclbin`
- z build procesu pomocí Vitis také do složky `transfer`.

Aby bylo možné využívat příkazy `xmutil`, které využívají DFX-MGR (daemon) [[xilinx-github-dfx-mgr](#)] pro konfiguraci PL, loading a unloading bitstreamů aplikací, spravování data modelu daných bitstreamů apod. je třeba pro akcelerovanou aplikaci vytvořit metadata soubor `shell.json`. Tento soubor je využíván právě DFX-MGR. Podpora dalších formátů `shell_type`, než jen `XRT_FLAT` je ve vývoji. [[xilinx-github-vitis-tutorials-step-2-create-the-software-components](#)]

Tento soubor je opět vhodné vytvořit na vývojářském PC a přemístit do složky `transfer`.

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

Kód 14 - 1 Metadata `shell.json` soubor pro `xmutil`.

- 15 Představení pracoviště**
- 16 Dosažené výsledky**

## Závěr

Aliquam dapibus leo velit, ultrices eleifend mi feugiat eget. Aliquam euismod facilisis turpis, nec lobortis libero aliquet sit amet. Aenean suscipit ante eget ipsum viverra hendrerit. Ut sed massa sed nisi tempus dapibus in eu enim. Nullam vitae odio laoreet, malesuada purus non, faucibus orci. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam eget odio quis enim laoreet imperdiet nec eu nunc. Maecenas ut consequat purus. Duis faucibus risus nec metus cursus placerat. Phasellus sapien justo, laoreet in pulvinar ut, maximus nec velit.

## **Příloha A: Seznam symbolů a zkratek**

### **A.1 Seznam symbolů**

$\vec{F}$  (N) vektor síly

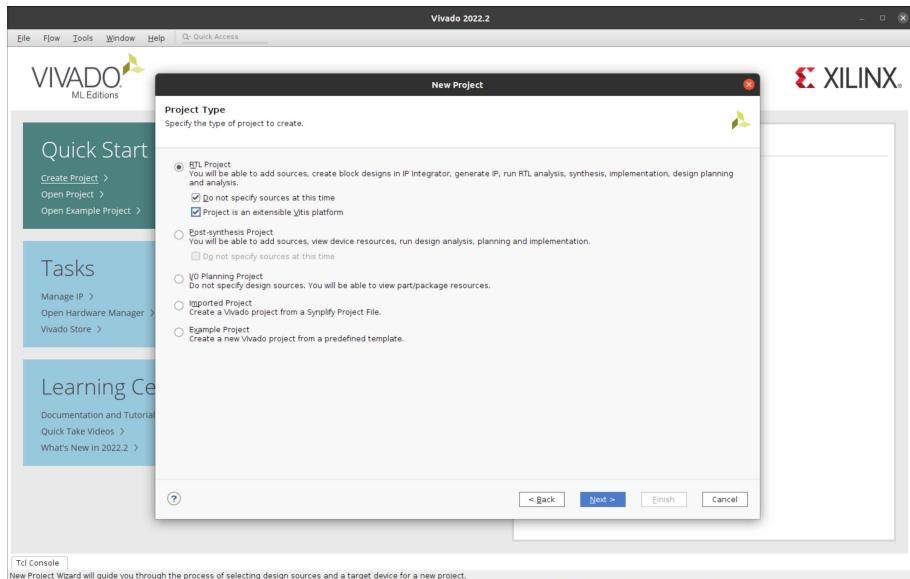
### **A.2 Seznam zkratek**

DCM DC Master

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

V této části bude představen postup tvorby HW architektury pro vývojovou desku Digilent Zybo Zynq-7000. Postup tvorby platformy byl částečně převzat z [hackster-vitis-2021-1-embedded-platform-for-zybo-z7-20]. V případě, že je požadováno vytvoření dalších speciálních bloků v čipu PL je třeba do blokového designu vložit odpovídající IP bloky, které zajistí potřebnou funkcionality.

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. ??.



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

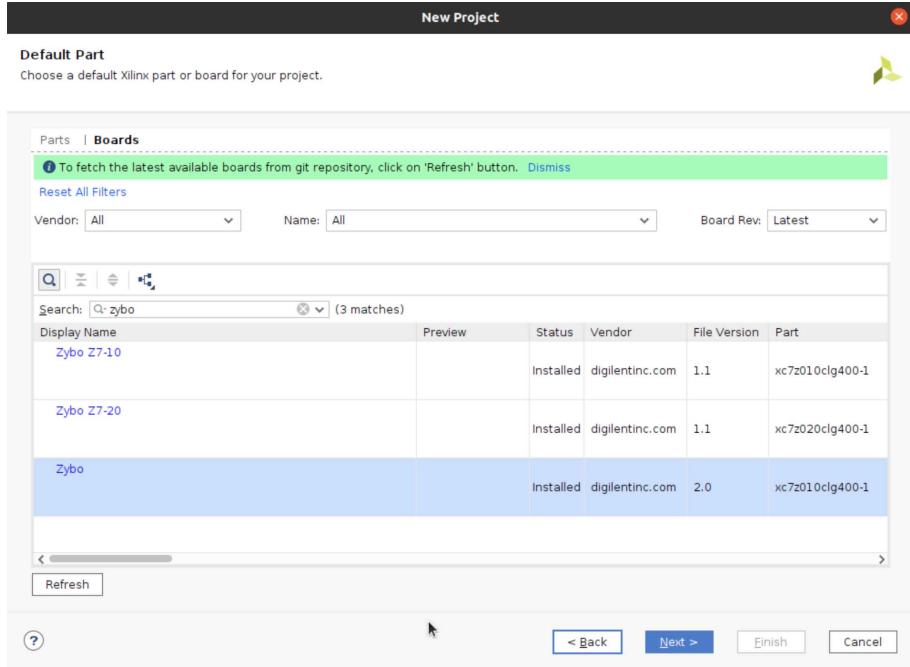
Následně v nabídce *Xilinx part* vybrat možnost *Board* a do vyhledávání zadat název využívané desky. V této práci bude využíváno desky *Zybo*. Díky instalovaným *board files*, představených v části *Vivado Board Files*, je možné nalézt požadovanou desku verze 2.0 a pokračovat v tvorbě designu. Výběr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

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. ??.

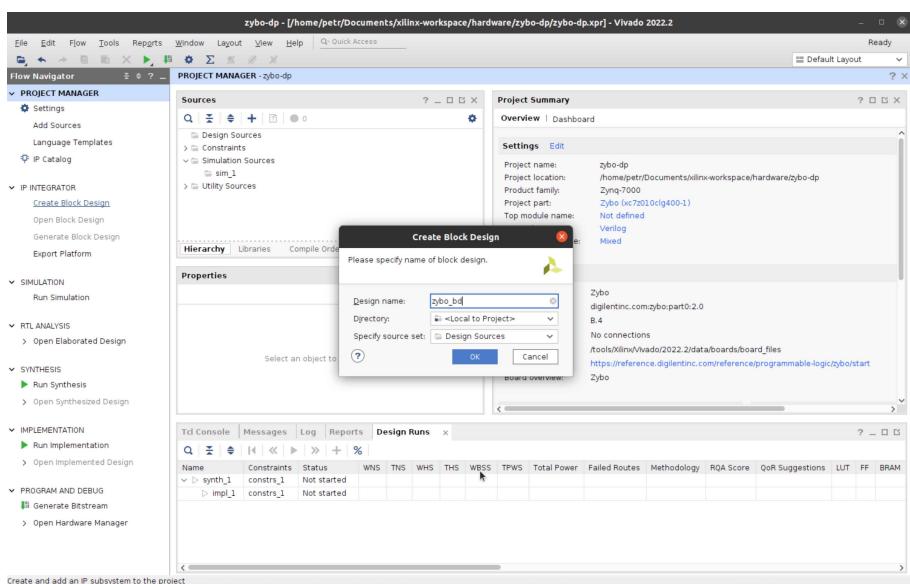
Nyní je možné již přistoupit k tvorbě vlastní architektury. Prvotním krokem tvorby fesignu je vložit blok *ZYNQ7 Processing System* a zvolit nově zobrazenou možnost *Run Block Automation*. V těchto pomocných automatizacích je většinou výhodné ponechávat nastavené výchozí hodnoty, které jsou pro většinu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

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 Hz. Poté je nutné na výstup *FCLK\_CLK0* bloku *ZYNQ7 Processing System* připojit vstup *clk\_in1* a k výstupu *FCLK\_RESET0\_N* vstup *resetn*.

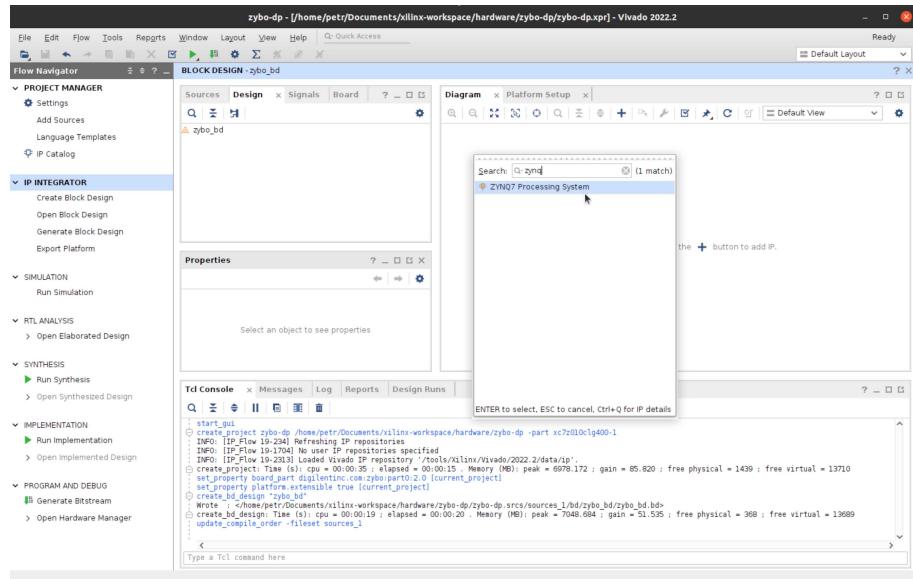
Po nastavení bloku *Clocking Wizard* je zapotřebí do designu vložit pět bloků *Processor System Reset*. Následuje propojení odpovídajících výstupů bloků *Clocking Wizard* s názvem *clk\_outX*, kde *X* značí po-



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

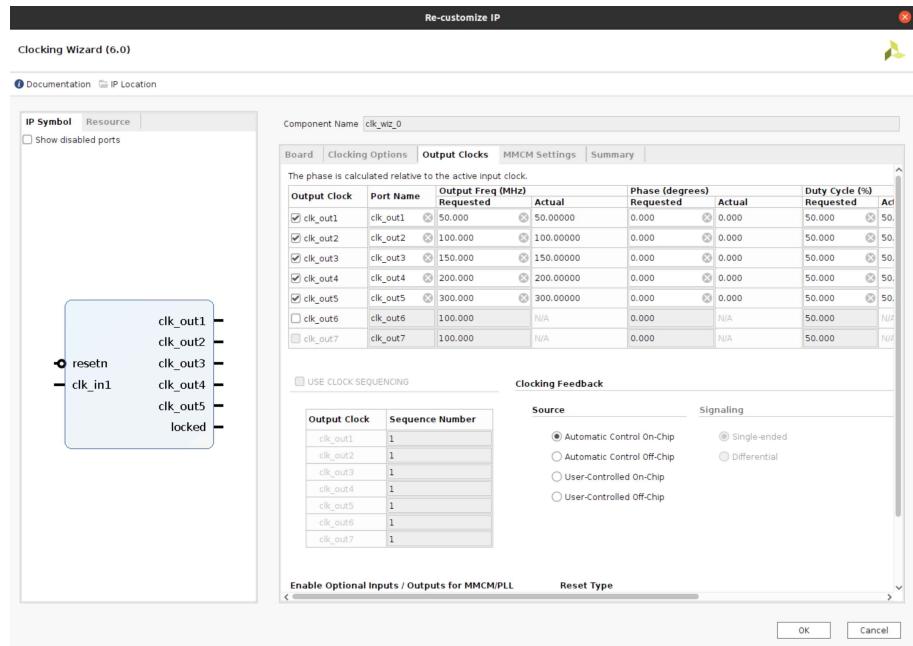


Obr. B - 3 Xilinx Vivado – vytváření Block Design pro Digilent Zybo.

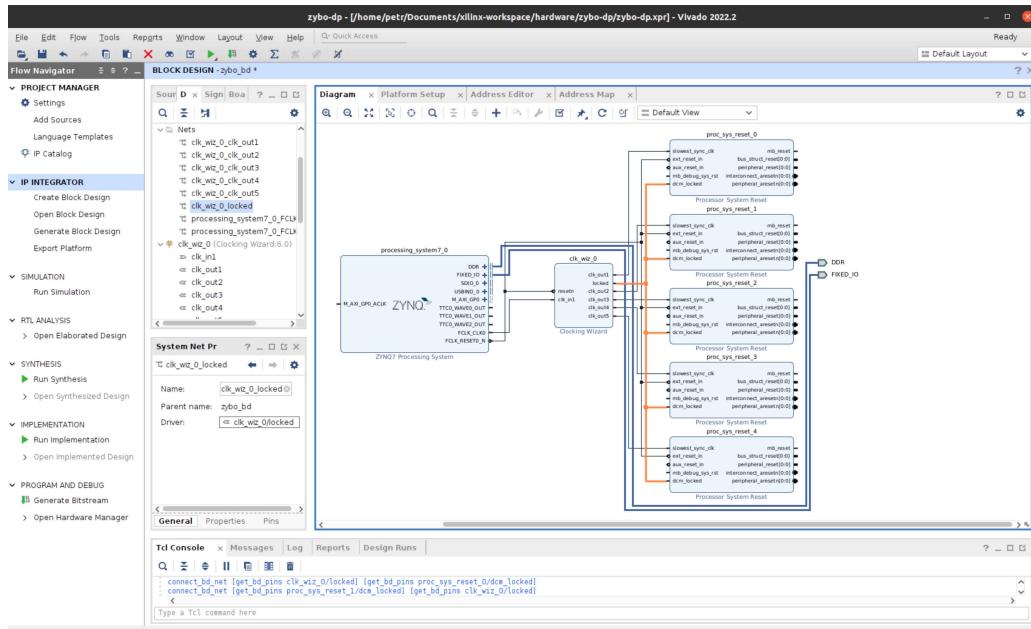


Obr. B - 4 Xilinx Vivado – vložení bloku ZYNQ7 Processing System pro Digilent Zybo.

řadí výstupního signálu, s odpovídajícím bloky *Processor System Reset* a jejich vstupy *slowest\_sync\_clk*. Ke všem vstupům *dcm\_locked* bloků *Processor System Reset* je nutné připojit výstup *locked Clocking Wizard*. A konečně ke všem vstupům *ext\_reset\_in* připojit výstup *FCLK\_RESET0\_N* ZynQ bloku. Představené propojení jednotlivých bloků je možné pozorovat na obr. ??.



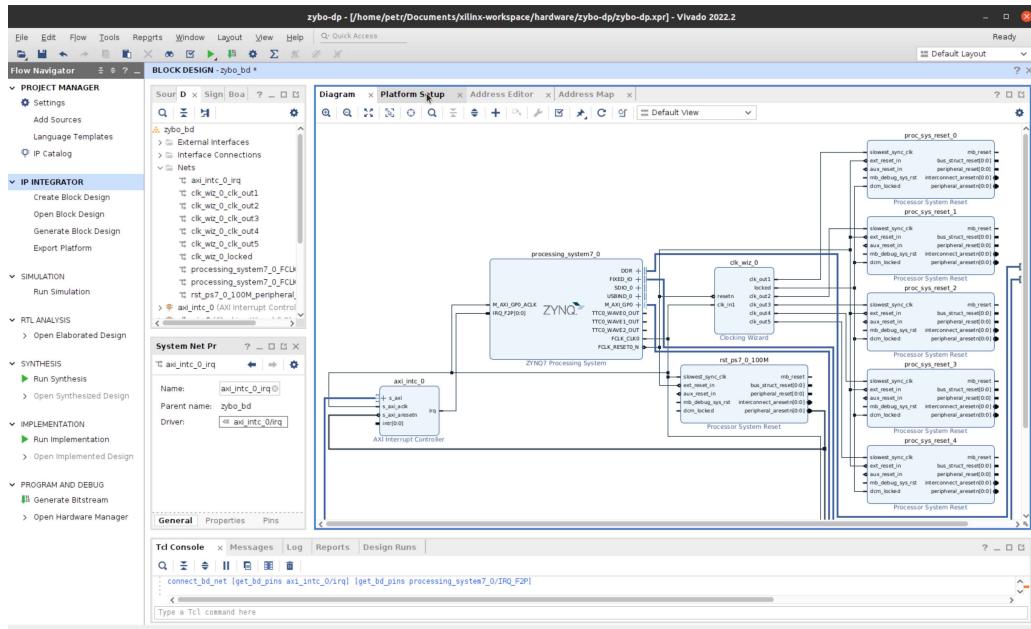
Obr. B - 5 Xilinx Vivado – nastavení výstupních taktovacích signálů pro Digilent Zybo.



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 se jménem *AXI Interrupt Controller*, otevřít jeho nastavení a v sekci *Processor Interrupt Type and Connection* změnit nastavení *Interrupt Output Connection* z *Bus* na *Single*. Následně je opět možné spustit automatické propojení jednotlivých bloků s výchozím nastavením.

Aby byly přerušení funkční, je třeba propojit výstup bloku *AXI Interrupt Controller* se jménem *irq* se vstupem bloku *ZYNQ7 Processing System IRQ\_F2P*. Minimální funkční blokový design je zobrazen na obr. ??.



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* ve které je pro zajištění funkč-

nosti nutné nastavit určité potřebné konektory a výstupy. Následuje nastavení parametrů bloku *ZYNQ7 Processing System* v záložce *AXI Port* dle tabulky č. ??.

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

| Name      | Enabled | Mexport  | 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. [\[hackster-vitis-2021-1-embedded-platform-for-zybo-z7-20\]](#)

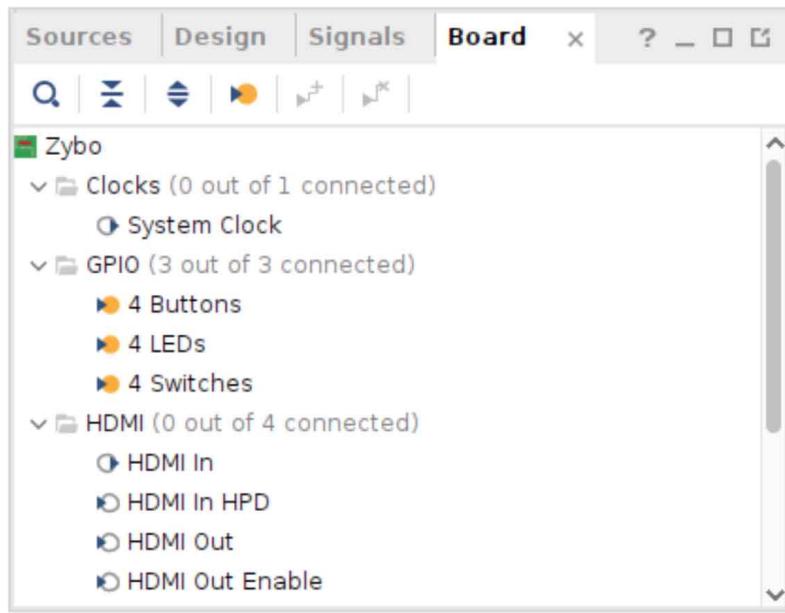
V této práci nebylo dosaženo funkční SW ani HW simulace pro desku Digilent Zybo z neznámých důvodů. Pokud byl emulátor QEMU spuštěn zvlášť pomocí příkazové řádky a přes příkaz byly *scp* přesunuty potřebné soubory, jednalo se pouze o SW simulaci bez připojeného PL a tudíž nebylo možné algoritmy ověřit.

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 blok je nutné zvolit z nabídky *Connect board component* a 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. ??). 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. ??.

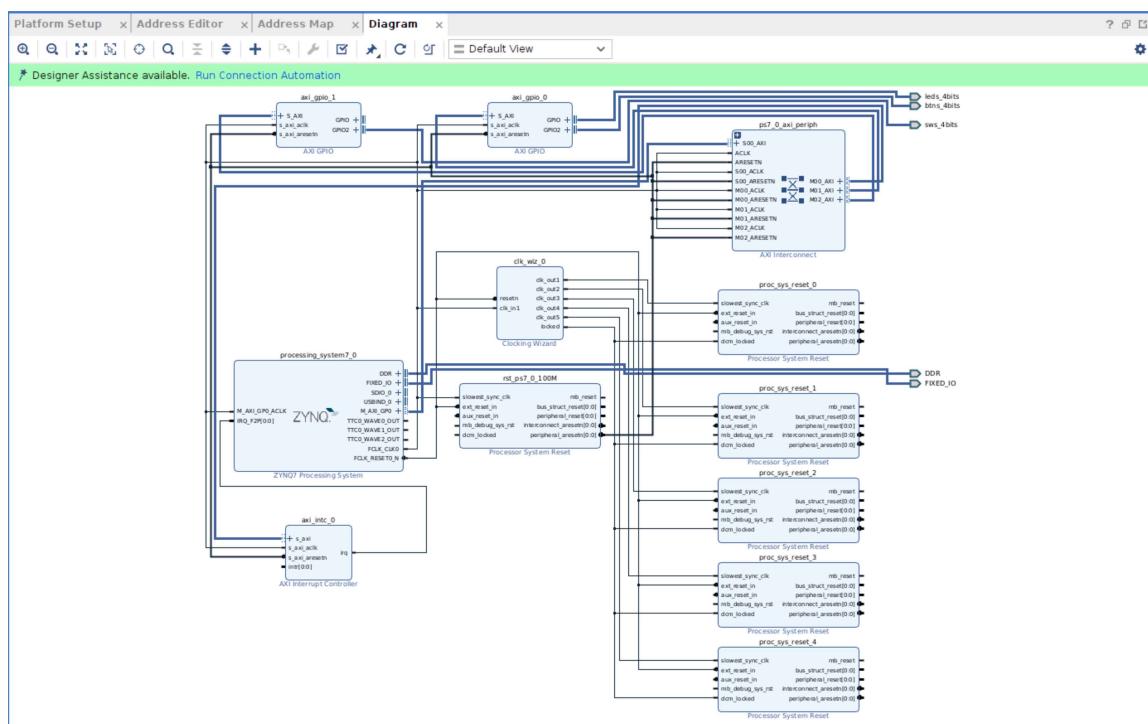
Rozdělení vstupů a výstupů do jednotlivých částí SoC – PS a PL je vyzobrazeno na obr. ??.

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 vyskytují kritická upozornění, která jsou zobrazena například na obr. ??, je stále možné pokračovat v tvorbě konečného produktu.

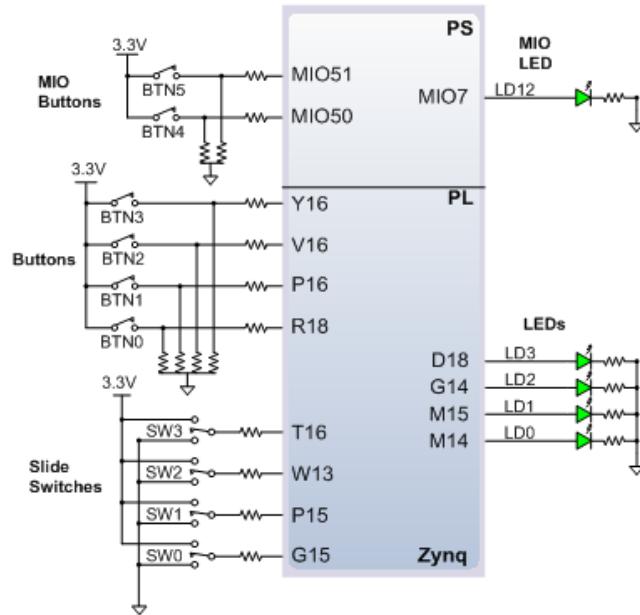
Po vytvoření *HDL Wrapper* je možné v menu *Flow Navigator/Program and Debug* zvolit krok *Generate Bitstream*. Pokud do tohoto kroku nebyla provedena syntéza ani implementace designu, objeví se hlášení, že je třeba tyto kroky provést, v případě pokračování v požadavku generování bitstreamu budou automaticky provedeny. V navazující nabídce možné vybrat, zda procesy budou probíhat lokálně či na



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

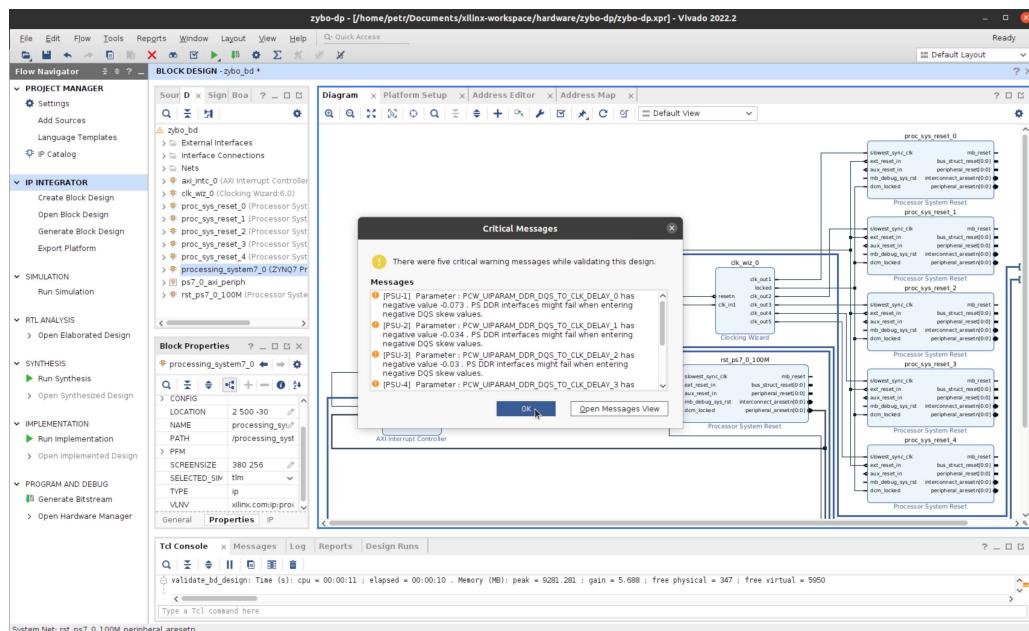


Obr. B - 9 Xilinx Vivado – block design s využitím GPIO pro LED, BTN, SW propojených s PL pro Digilent Zybo.



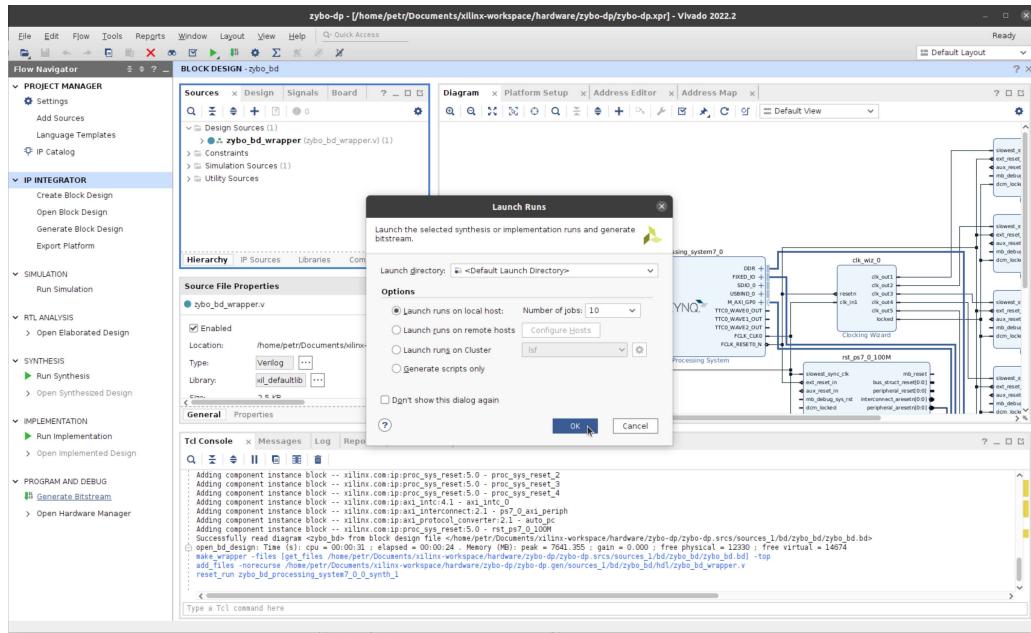
Obr. B - 10 Uspořádání připojení tlačítek, přepínačů a LED k PS a PL pro vývojovou desku Digilent Zybo. [\[digilent-zybo-reference-manual\]](#)

vzdáleném serveru, nebo clusteru. Také je možné zvolit kolik výpočetních jader procesoru 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 a proces syntézy, implementace a generace bitstreamu bude nutné spustit znova. Ukázka nastavení procesu je zobrazena na obr. ??.



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

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



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.

procesů je umístěn v kartě *Log*.

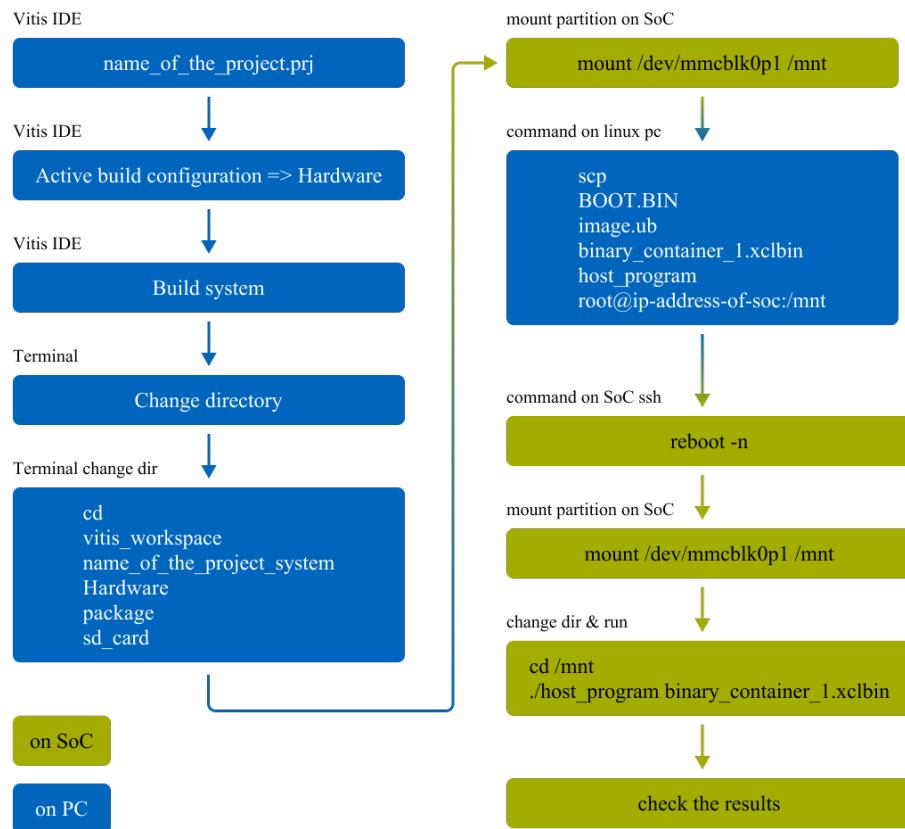
Po úspěšném provedení jednotlivých kroků je zobrazena nabídka, která umožňuje nahlédnout na vytvořený design. Tuto nabídku je možné zavřít aniž by byla vykonávána jakákoli z nabízených možností.

Po ukončení procesů je pro možné použití vytvořeného designu pro tvorbu PetaLinux systému a aplikací v Xilinx Vitis nutné exportovat vytvořenou platformu. Export je proveden pomocí sekvence tlačítek *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 aplikace v Xilinx Vitis pomocí C++. Důležitou volbou je zvolení možnost *Include bistream*, který byl produktem tvorby designu v této části.

Po nastavení dodatečných informací platformy je možné jej vyexportovat do požadované lokace, kde bude dále využívána.

## Příloha C: Upravený postup debuggingu PL pro Digilent Zybo

Debugging pro program pro PL. Pro host program je možné debuggovat přímo ve Vitis, ale bez PL kernelu.



Obr. C - 1 Diagram popisující upravený postup pro debuggování a spouštění host programu a nahrávání host programu do PL.