VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EMULACE PERIFERIÍ VESTAVĚNÝCH SYSTÉMŮ PRO RYCHLÉ PROTOTYPOVÁNÍ

EMBEDDED SYSTEM PERIPHELARS EMULATION FOR FAST PROTOTYPING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DOMINIK MÜLLER**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VOJTĚCH MRÁZEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Müller Dominik, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Vestavěné systémy
Název: **Emulace periferií vestavěných systémů pro rychlé prototypování**
Embedded System Peripherals Emulation for Fast Prototyping
Kategorie: Vestavěné systémy
Zadání:

1. Seznamte se s možnostmi prototypování a automatického testování vestavěných systémů.
2. Prozkoumejte možnosti emulace vybraných periférií s využitím obvodů FPGA.
3. Zpracujte rešerši na výše uvedená témata.
4. Navrhněte systém umožňující emulovat vnější periferie vestavěných zařízení v FPGA.
5. Navržený systém implementujte.
6. Vyhodnoťte jeho vlastnosti a konfigurovatelnost a diskutujte možnosti dalšího rozšíření.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Mrázek Vojtěch, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 29. října 2021

Abstrakt

Tato práce se zabývá návrhem a konstrukcí vývojové a testovací platformy pro vývojáře vestavěných systémů určené k akceleraci počátečních fází vývoje vestavěných systémů. Vytvořená platforma umožňuje vývojáři emulovat real-time okolí vyvíjeného systému, které může pozorovat a ovlivňovat. Návrh celé platformy se soustředil na její rozsáhlou konfigurovatelnost, snadnou rozšířitelnost, znovupoužitelnost a univerzalitu. Simulace probíhá přímo vůči reálnému mikrokontroléru. Platforma tak doplňuje přístup čistě softwarové simulace o reálný základ, ale zůstává, oproti specializovaným testovacím systémům, znovupoužitelná a cenově dostupná. Výsledkem práce je fyzické zařízení ovladatelné přes počítač uživatele umožňující připojit vývojový kit a simulovat jeho okolí.

Abstract

This thesis deals with the design and implementation of a development platfrom for embedded system developers intended to accelerate initial phases of development cycle. The proposed platform allows to emulate real-time environment of the system under development with possibility to observe and change the environment at runtime. The design of the entire platform focused on its extensive configurability, ease of extensibility, reusability and versatility. Simulation is performed directly against a real microcontroller. The platform thus complements the pure software simulation approach with a real-world basis, but remains reusable and affordable compared to dedicated test systems. The result of this work is a physical device controllable via the user's computer allowing to connect the development kit and simulate its environment.

Klíčová slova

Hardware-in-the-loop, Processor-in-the-loop, Real-time Simulace, Akcelerátor vývoje vestavěných systémů

Keywords

Hardware-in-the-loop, Processor-in-the-loop, Real-time Simulation, Embedded System Devolopment Accelerator

Citace

MÜLLER, Dominik. *Emulace periferií vestavěných systémů pro rychlé prototypování.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch Mrázek, Ph.D.

Emulace periferií vestavěných systémů pro rychlé prototypování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Vojěcha Mrázka PhD. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Dominik Müller
18. května 2022

Poděkování

Děkuji panu Ing. Vojtěchovi Mrázkovi, Ph.D., za vedení, konzultace a cenné poznatky při vypracování této práce.

Obsah

1 Úvod **3**

2 Vestavěný systém **5**
2.1 Druhy vestavěných systémů 6
2.2 Vývoj vestavěných systémů 8
2.3 Simulace částí vestavěných systémů 10
2.4 Emulace hardwarových periférii na FPGA 11
2.5 Testování vestavěných systémů 12

3 Rozhraní **15**
3.1 Rozhraní uvnitř rozhodovacího prvku 15
3.2 Rozhraní vůči desce plošných spojů 16
3.3 Rozhraní vůči okolí 22

4 Popis problému **23**
4.1 Existující řešení 24
4.2 Stanovení požadavků navrhované platformy 26

5 Návrh **28**
5.1 Platforma 28
5.2 Simulace prostředí testovaného systému 33
5.3 Softwarové rozhraní navržené platformy 34
5.4 Firmware RTOS jádra 37
5.5 Periférie v FPGA 37
5.6 Fyzická část platformy 42

6 Realizace **50**
6.1 Vytvoření bitstreamu 50
6.2 Sestavení systémů platformy 57
6.3 Aplikace uvnitř platformy 60
6.4 Implementace simulačního prostředí 62
6.5 Fyzická podoba platformy 62

7 Vyhodnocení **64**
7.1 Ukázky 64
7.2 Vyhodnocení požadavků na platformu 67
7.3 Budoucí rozšíření 68

8 Závěr **70**

Literatura **71**

A Schéma hlavní desky platformy **73**

B Schéma adaptéru **79**

C Blokové schéma návrhu uvnitř FPGA **81**

Kapitola 1

Úvod

Vestavěné systémy se vyskytují všude kolem nás a staly se nezbytnou a užitečnou součástí našich životů. Tyto systémy představují jednoúčelová zařízení komunikující s vnějším prostředím za použití elektronických analogových a digitálních signálů. Jako příklady lze uvést např. řídící systém robota, různé druhy směrovačů nebo elektronický termostat. Jejich komplexnost však může komplikovat vývoj a testování firmware. Například při paralelním vývoji několika částí systému potřebujeme nahradit chybějící části funkčně podobným či ekvivalentním protikusem. Takovéto problémy typicky řešíme simulací některých částí. Tento přístup se používá také pro testování zařízení, která by při testování mohla svým nesprávným chováním způsobit újmu na zdraví, financích nebo poškodit životní prostředí (např. hlásič požáru, naváděcí systém rakety, ...). V rámci vývoje využíváme simulaci také pro navození zřídka vyskytovaných jevů nebo k urychlení dlouho probíhajících dějů (např. porucha, ohřívání, ...).

Nástroje pro simulaci existují a jsou běžně využívány. Čistě softwarová simulace však vyžaduje velkou míru aproximace, při které vznikají nepřesnosti. Na druhou stranu běžně využívané testovací systémy hardware-in-the-loop, které simulují pouze okolí testovaného zařízení, jsou komplexní a finančně náročné. Z tohoto důvodu byla navržena speciální platforma, která bude umožňovat snadnou a rychlou simulaci vnějšího prostředí vestavěného systému za přijatelnou cenu. Platforma si neklade za cíl nahradit dostupné nástroje, nýbrž představit odlišný pohled a společně s existujícími nástroji vylepšit vývojový proces. Cílem navrhované platformy je vytvořit pro uživatele snadno konfigurovatelné a rozšiřitelné okolí procesoru, se kterým může interagovat. A to takovým způsobem, aby mohl ověřit funkčnost či realizovatelnost své implementace na fyzické úrovni přímo s vybraným procesorem a bez zásahu do způsobu vývoje na tomto procesoru. Dalším cílem platformy je spojení nejčastěji používaného vývojového vybavení do jednoho produktu. A to tak, aby ji mohl mít každý vývojář u sebe na stole a mohl ji snadno a rychle začlenit do svého pracovního procesu.

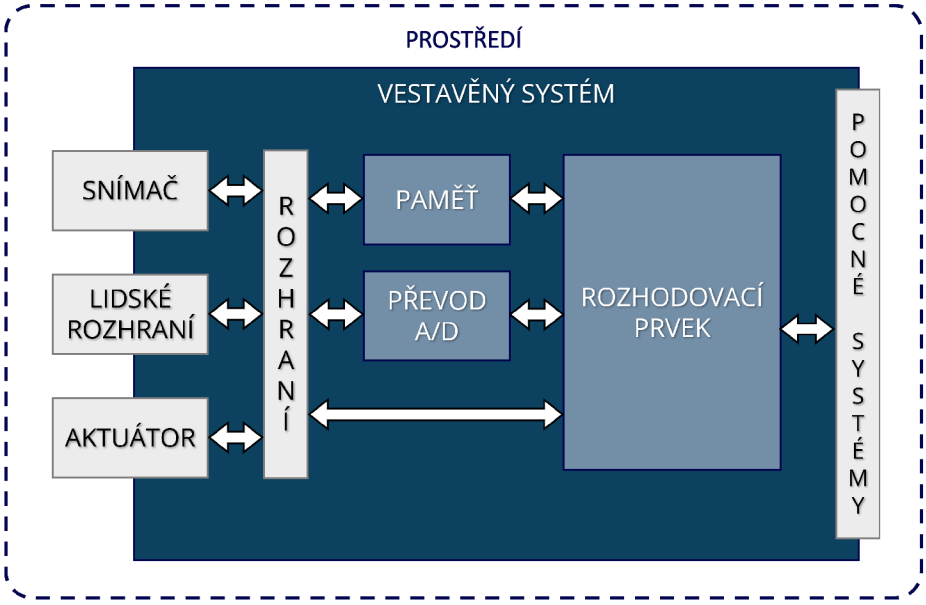
Mnou navrhovaná platforma umožňuje její použití ve všech fázích vývoje vestavěného zařízení. Na počátku vývoje slouží platforma jako pomocník při evaluaci — spouštění benchmarků, průzkumu práce s periferiemi atd. Po specifikaci požadavků umožňuje platforma simulaci zařízení v okolí mikrokontroléru a umožňuje tak vývojáři pohodlný vývoj a testovaní do doby, než má k dispozici první prototyp. Po obdržení fyzického prototypu slouží platforma vývojáři převážně pro testování hraničních situací, ladění chyb a jako základ pro automatické testování HIL. Díky tomuto můžeme i ve vestavěném zařízení a jeho fyzickém rozhraní využívat koncepty známé z klasického softwarového vývoje, jako jsou automatické unit-testy a podobně. Tyto testy mohou být automatizovány pro všechny další případné revize vyvíjeného systému.

Práce začíná rešeršní částí, ve které se čtenář může seznámit s probíranou problematikou. Tato část by měla odpovědět na otázky: Co je to vestavěný systém? (kapitola 2), Jak se takový systém vyvíjí a testuje? (sekce 2.2), S jakým rozhraním se můžeme setkat a jak ho můžeme emulovat? (kapitola 3). V kapitole 4 jsou vysvětleny motivace a cíle pro navrhovanou platformu a jsou popsána existující řešení, která alespoň z části umožňují splnit stanovené motivace. Existující řešení jsou vyhodnocena jako nevyhovující a v kapitole 5 je představen návrh nové platformy. V této kapitole je stanoveno z jakých částí se platforma skládá a tyto části jsou následně detailněji rozebrány. Z představeného návrhu je vytvořen finální produkt. Průběh jeho implementace je rozebrán v kapitole 6. Ta obsahuje popis prostředí, ve kterém byla platforma vyvíjena a motivace pro použití konkrétních nástrojů. Kromě popisu prostředí kapitola obsahuje také konkrétní implementační detaily jednotlivých částí platformy, popis jejich chování a případné odlišnosti od původního návrhu. Na konci kapitoly pak lze vidět finální systém s popisem jak ho zprovoznit. Po implementaci přichází finální kapitola 7, ve které je shrnuto, co vše platforma dokáže a jaké jsou její limity. Dále jsou také uvedeny ukázky z prostředí platfromy, ve kterých lze získat přehled o tom, jak se s platformou pracuje. Konečně je také vyhodnoceno naplnění požadavků na platformu, stanovených v popisu problému.

Kapitola 2

Vestavěný systém

Vestavěný systém je obecný pojem, který označuje kombinaci hardwaru a softwaru, jejímž smyslem je řídit externí proces, zařízení nebo systém a který je vestavěný do uzavřeného produktu [24]. Vestavěný systém tak můžeme nalézt v široké škále zařízení, od mluvící panenky pro děti až po naváděcí systém rakety. Obrázek 2.1 zobrazuje obecný diagram aplikovatelný pro většinu vestavěných systémů.



Obrázek 2.1: Obecný diagram vestavěného systému.

Vestavěné systémy jsou typicky vyvíjeny pro manipulaci svého prostředí. Získávají informace o stavu prostředí ze senzorů a ovlivňují ho pomocí aktuátorů. Příslušnou reakci systému definuje kód systému uložený v paměti. Jelikož je prostředím vestavěného systému reálný svět, často nás zajímají spojité signály (např. rychlost, teplota). Digitální systém tyto signály nedokáže zpracovat a je tak zapotřebí převodník. Dle případu užití systému pak záleží na tom, je-li převod realizován v jednom, či obou směrech — analogově digitální převodník (ADC) nebo digitálně analogový převodník (DAC). Celý systém také potřebuje k provozu pomocné systémy. Minimálně zdroj energie, ten může získat buď externě, nebo vlastními zdroji (např. baterie). [9]

Vestavěné systémy můžeme dále přesněji kategorizovat jako:

- systémy pro zpracování signálu,
- kritické systémy,
- distribuované systémy,
- malá spotřební elektronika.

Každá kategorie má odlišné požadavky, nároky na hardware i přístup k vývoji produktu. [18]

2.1 Druhy vestavěných systémů

Kritický systém

Kritický systém je takový systém, jehož chyba by mohla vést k situacím, které jsou neakceptovatelné. Mezi tyto situace můžeme zařadit například ohrožení na životě, způsobení značných ekonomických ztrát nebo rozsáhlé škody na životním prostředí. [13] Takovéto systémy tak nesmí selhat vůbec (předejít selhání) nebo musí zaručit, že selžou bezpečně.

Systémy pracující v reálném čase

Vestavěné systémy často pracují v reálném čase (real-time). Real-time systém je takový systém, který musí pro svoji správnou funkci naplnit časová omezení. Vestavěné systémy mohou, ale nemusí, mít časové omezení [25]. Rozlišujeme také závažnost jejich porušení. Pokud se jedná např. o zpracování videozáznamu, potřebujeme snímek zpracovat předtím, než přijde další. Pokud toto omezení porušíme, systém funguje dále a uživatel to nemusí vůbec postřehnout. Na druhou stranu systém pro železniční přejezd musí nutně zaručit sklopení závory před příjezdem vlaku. Rozlišujeme tedy systémy:

- **Hard real-time systém** musí vždy splnit časová omezení, protože jejich nesplnění, by mohlo vést k neakceptovatelné situaci [19].
- **Soft real-time systémy** jsou takové real-time systémy, které nejsou hard [19].

Rozhodovací prvek

Rozhodovací prvek vestavěného systému je jeho nejdůležitější částí. Jedná se o část, která nejvíce ovlivňuje vývoj produktu a spojuje celý systém dohromady. Jeho úkolem je manipulace dat (informace) takovým způsobem, aby systém splňoval definované požadavky. Pro řízení systému nejčastěji volíme:

- Mikrokontrolér (MCU),
- Mikroprocesor (MPU),
- Digitální signálový procesor (DSP),
- Rekonfigurovatelné systémy - FPGA,
- Specializované obvody - ASIC,
- Systém na čipu (SoC).

Mikrokontrolér

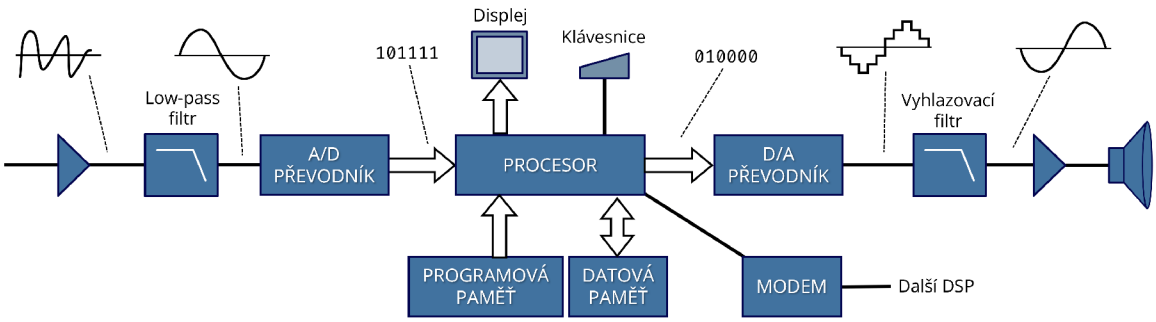
Mikrokontrolér je samostatný systém s jedním či více jádry procesorů, pamětí a periferiemi distribuovaný v jednom pouzdře [12]. Bývá navržený tak, aby ke své funkci vyžadoval co nejméně externích komponent. Periférie uvnitř mikrokontroléru jsou speciální pomocné obvody plnící určitou funkci. Periférie mikrokontroléru volí výrobce a nejčastěji se jedná o periférie pro komunikaci s okolím, časovače nebo akcelerátory. Mikrokontroléry jsou obvykle zdrojově omezený systém. Velikost pamětí se zřídkakdy pohybuje v desítkách megabytů.

Mikroprocessor

Mikroprocesor je samostatný systém, s jedním či více procesory, distribuovaný v jednom pouzdře. Potřebná paměť a další periférie jsou dodány externími součástkami. [12] Samotný procesor je oproti mikrokontroléru obvykle výkonnější a také více vhodný pro nasazení operačního systému (Linux).

Digitální signálový procesor

Digitální signálový procesor je procesor speciálně navržený pro zpracování digitálních signálů. Jedná se o zpracování signálu pomocí digitálního hardwaru nebo nějakého výpočetního prvku. Ukázka běžného zpracování signálu je na obrázku 2.2. [10]



Obrázek 2.2: Obecný diagram DSP [10].

Oproti běžně používaným procesorům je hlavní rozdíl v architektuře. DSP využívají spíše harwardskou architekturu a mají více systémových sběrnic. [10]

Rekonfigurovatelné systémy – FPGA

V mnoha případech je softwarové řešení příliš pomalé nebo příliš energeticky náročné. Rekonfigurovatelné systémy umožní implementovat efektivnější řešení v konfigurovatelném hradlovém poli a přitom zachovají možnost implementaci později nadále upravovat. Field programmable gate arrays (FPGA) jsou nejčastější instancí rekonfigurovatelného systému. FPGA kromě konfigurovatelného regionu obsahuje také I/O rozhraní, specializované obvody (např. násobička) a paměť RAM. [19]

Konfigurovatelnost je docílená pomocí konfigurovatelných logických bloků (CLB), které se skládají z paměti RAM, registrů a multiplexorů. Paměť RAM využíváme jako look-up tabulku (LUT), kde každá buňka tabulky obsahuje výstupní hodnotu (0 nebo 1) a kombinací vstupů tyto buňky indexujeme. Jsme tak schopni implementovat libovolnou booleovskou

funkci až do k proměnných, kde k je počet vstupů LUT. Pomocí jednotlivých CLB tvoříme sčítačky, multiplexory, posuvné registry nebo paměti. Skládáním těchto primitiv dále tvoříme složitější objekty a algoritmy. [19]

Rekonfigurovatelné systémy jsou z implementačního hlediska oproti procesorům na nižší úrovni. Jejich chování implementujeme v jazycích popisující hardware. Pomocí rekonfigurovatelných systémů tak lze implementovat i jádro procesoru (např. Cortex M1). Nižší úroveň nám přináší větší volnost při návrhu systému, ale za cenu větší komplexity vývoje.

Systém na čipu

Systém na čipu (SoC) je samostatný systém kombinující více výše popsaných systémů dohromady. Výhodou těchto systémů je, že umožňují větší flexibilitu a zároveň výrobce implementuje propojení mezi systémy. Tím, že jsou všechny systémy v jednom pouzdře, je možné dosáhnout větší propustnosti při zabrání méně místa, než by tomu bylo v případě vlastní implementace. [8]

Můžeme se tak setkat s SoC kombinujícími mikroprocesor a FPGA, mikroprocesor a DSP a další. Takovéto systémy nám umožňují lépe rozložit jednotlivé implementační bloky mezi hardware a software a dosáhnout tak efektivnější implementace. Např. pro SoC CPU+FPGA můžeme v CPU implementovat řízení a vyhodnocení a v FPGA samotné zpracování signálu s případnou akcelerací části vyhodnocovacího algoritmu.

2.2 Vývoj vestavěných systémů

Z definice vestavěného systému vyplývá, že vývoj se týká jak softwaru, tak i hardwaru. Software je tak velmi svázaný s výsledným hardware a zvolený rozhodovací prvek značně ovlivňuje další vývoj. Využití rekonfigurovatelného hardware vyžaduje naprosto odlišný přístup od vývoje mikrokontroléru. Rozdíly, ačkoliv ne tak znatelné, jsou i mezi rozhodovacími prvky ve stejné kategorii. Například dva mikrokontroléry mohou implementovat tu samou funkci, jeden ji však implementuje v software a druhý pro ni má hardware periférii. Podobný problém může nastat i mimo rozhodovací prvek. Mějme například dva senzory měřící teplotu. Z hlediska funkce jsou ekvivalentní, ale každý má rozdílné rozhraní. Takovéto problémy se obvykle snažíme minimalizovat vytvořením abstrakce hardwarové úrovně tak, aby případná změna hardware neovlivnila aplikační logiku. Do jisté míry tuto abstrakní vrstvu vytváří již výrobce, aby usnadnil vývoj a zvýšil atraktivnost platformy.

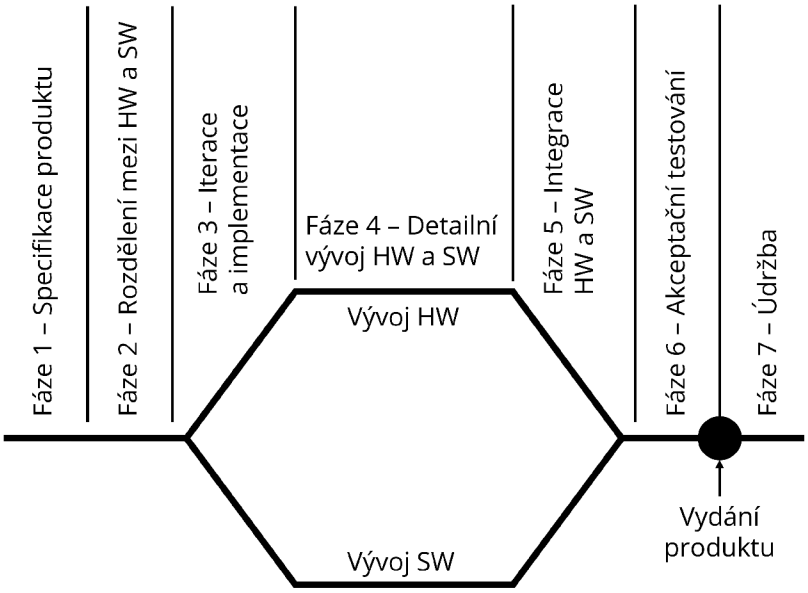
Obrázek 2.3 ukazuje standardní fáze vývoje vestavěného produktu. Průběh však není přímočarý a v praxi nastane nemalý počet iterací a optimalizací mezi fázemi.

Specifikace požadavků

Na počátku vývoje musí vzniknout seznam požadavků, které musí výsledné zařízení splnit. Tento krok je velmi důležitý, protože neúplná nebo nesprávná specifikace vede k problémům v dalších fázích vývoje. Podle těchto požadavků se také odvíjí rozhodnutí v dalších částech vývoje. [8]

Rozdělení hardwaru a softwaru

Jak již bylo zmíněno, vestavěný systém se skládá z hardwaru a softwaru. Během vývoje tak musíme rozhodnout, kde bude jaká funkcionalita implementována. Rozdělení je komplexní optimalizační problém a je typicky ovlivněno cenou a požadovaným výkonem. Software



Obrázek 2.3: Diagram životního cyklu vývoje vestavěného systému [8].

řešení je levnější, ale méně výkonné. Naopak hardware řešení je dražší, hlavně kvůli nákladnějšímu a delšímu vývoji, avšak je výkonnější. [8]

Iterace a implementace

V této fázi je návrh stále nestabilní a hardware a software cesty se začínají pomalu rozcházet. Už v této fázi by měly být rozhodnuté hlavní části návrhu, jako např. použité technologie. Jak hardware tým, tak software tým ověřují navržené rozdělení. Software tým ověřuje vhodnost zvolené platformy a provádí výkonnostní testy. Často k tomu využívá evaluačních kitů. Hardware tým provádí simulace a průzkum vhodných součástek. [8]

Návrh

Oba týmy pracují převážně odděleně na svých částech tak, jak si specifikovaly dříve. Hardware tým se snaží navrhnout první funkční prototyp. Software tým připravuje algoritmy a aplikační logiku. Jelikož software tým nemá k dispozici celý systém, může kompletně navrhnout pouze aplikační logiku. Části závislé na hardwaru může navrhnout pouze podle specifikace. Často se tak vytváří/využívají různé simulační nástroje pro známé hlavní komponenty a používají se evaluační kity.

Integrace

Integrační fáze nastává, když má hardware tým první funkční prototyp a software tým může začít produkt "oživovat". V této fázi se mohou objevit první neshody mezi hardware a software. Ideální stav je, pokud prototyp nevyžaduje žádné úpravy a software funguje na první pokus. Tento stav však reálně téměř nikdy nenastane. [8]

Testování

Míra testování je silně závislá od vyvíjeného produktu. Liší se jak způsob testování, tak i důslednost testů. Míra testování kritického systému bude značně odlišná od toho nekritického. Každý systém by ale měl splňovat akceptační testování. Tedy, že produkt splňuje všechny požadavky nastavené v úvodní fázi. [8] Více si tuto část rozebereme v sekci 2.5.

Údržba

Po vydání produktu nastává fáze údržby produktu. Pokud již sestavený systém nelze aktualizovat, týkají se změny hlavně kritických oprav nebo tvorby nových revizí při výpadku dodávky použitých součástek. V opačném případě může vývoj produktu pokračovat a mohou se objevovat i nově přidané funkce.

Prototypování

Prototypování je překlopení idey do funkční fyzické podoby. Výsledek prototypu nemusí ani zdaleka připomínat finální produkt. Cílem je ověřit hlavní myšlenku, se kterou prototyp vytváříme a objevit případné problémy dříve, než do nich vložíme větší úsilí. Během vývoje je vhodné vytvořit více prototypů a inkrementálně je přibližovat k výslednému produktu.

Vývoj vestavěných systémů bez vývoje předchozích modelů obvykle začíná v tomto bodě. Při prototypování postupně nahrazujeme původně simulované komponenty reálnými až do té doby, dokud nenahradíme všechny. [18]

2.3 Simulace částí vestavěných systémů

Simulace částí vestavěných systému nám umožňuje dříve dosáhnout funkčního prototypu a vyvinout tak systém rychleji a efektivněji. Během vývoje využíváme více přístupů k simulaci podle toho, který je pro nás v danou chvíli nejvhodnější.

Model-in-the-loop

Na této úrovni jsou vytvořeny modely pro všechny komponenty systému, je tak simulován celý systém. Jednotlivé modely jsou pouze prototypy a jejich podoba nemusí reflektovat tu finální. Obecně tato simulace slouží pro ověření konceptu. Můžeme tak např. ověřit že navrhovaný algoritmus splňuje očekávání, nebo že stavový automat řídící chování systému nevede k nevalidní situaci.

Software-in-the-loop

Při software-in-the-loop (SIL) ověřujeme chování vyvíjeného systému mimo jeho finální prostředí. K testu tedy stále nepotřebujeme žádný hardware, ale v simulaci už je použitý kód nasazený ve výsledném systému a v potaz je brána konkrétní podoba vyvíjeného systému [20]. Obvykle se pouští v prostředí vývojáře nebo automaticky při změně zdrojového kódu. Pro simulaci můžeme buď zvolit některý z dostupných emulátorů (např. QEMU, Renode) nebo konkrétní nuance použité platformy zanedbat a vytvořit abstrakní vrstvu. Ta by se navázala na nízkoúrovňové volání a nahradila implementační detaily vyvíjeného systému tak, aby umožnila průběh simulace.

Processor-in-the-loop

Processor-in-the-loop (PIL) je simulace okolí vestavěného systému, která není real-time, ale výsledný kód systému je vykonáván na konkrétním mikrokontroléru [26]. Sníží se tak nepřesnost chování oproti SIL, stále se ale zachovává flexibilita softwarové simulace.

Hardware-in-the-loop

Hardware-in-the-loop (HIL) je technika pro exekuci testů na úrovni celého systému rozsáhlým a opakovatelným způsobem [14]. Testuje se tedy bez jakékoliv aproximace na úrovni systému a simuluje se pouze jeho okolí, které musí být schopno reagovat v reálném čase. Simulace monitoruje výstupy testovaného systému a podle testovacího scénáře ovlivňuje jeho vstupy. HIL simulace probíhá v reálném čase, je tedy nutné dostatečně výkonné simulační prostředí. Z tohoto důvodu jsou často HIL systémy komplexní a úzce zaměřené na konkrétní vyvíjený systém. I nepatrná změna vyvíjeného systému tak může vyžadovat komplexní změnu nebo úpravu simulačního prostředí [26].

2.4 Emulace hardwarových periférii na FPGA

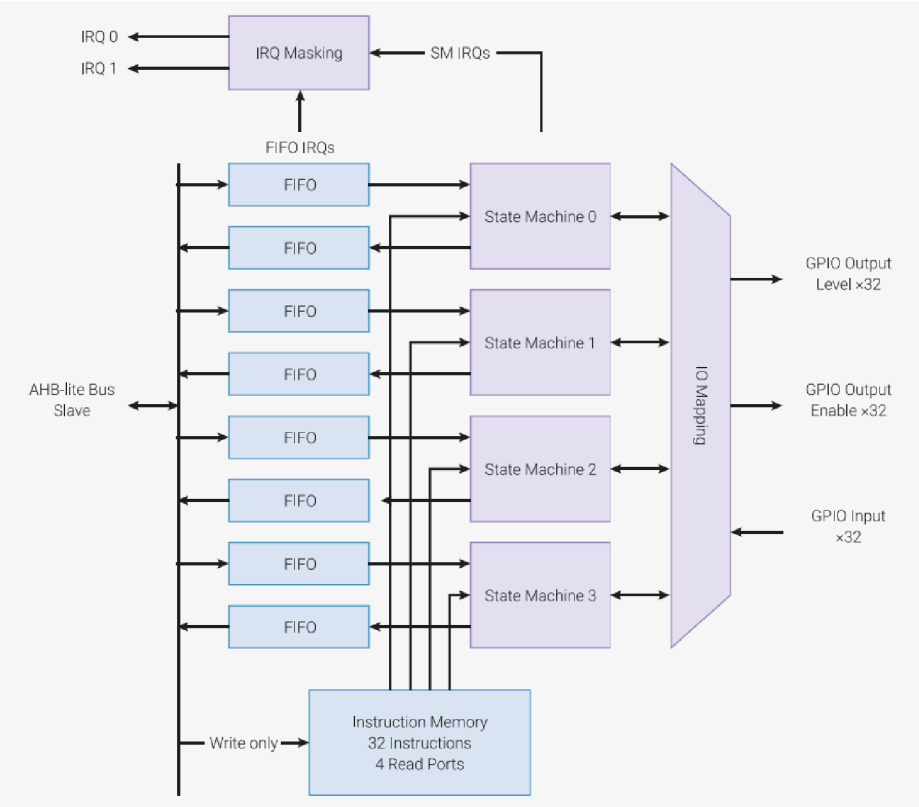
Emulace znamená napodobení chování nějakého systému na úplně jiném systému. Využíváme k tomu emulátor, jehož funkcí je vzít vstup a na jiném systému vytvořit takový výstup, který by odpovídal výstupu emulovaného systému.

Emulovat periférie na FPGA můžeme tak, že je popíšeme pomocí jazyka pro popis hardware (VHDL, Verilog atd.). Skládáním logických funkcí tak poskládáme celý emulovaný systém. Druhý způsob je vytvořit řídící prvek, který přijímá instrukce tvořící chování emulovaného systému. Vytvoříme tak speciální procesor pro emulaci daného systému. Výhodou takového systému je, že zachovává jistou míru flexibility a jednou hardware jednotkou můžeme pomocí více programů emulovat hned několik systémů. Nevýhodou pak je, že kvůli přidané režii je náročnější na zdroje a zároveň neumožňuje dosáhnout takových rychlostí jako periférie navržená na míru. Poslední možností je obecné jádro procesoru implementované uvnitř FPGA. Tato jádra jsou dostupná buď přímo od výrobce FPGA (např. Microblaze) nebo od třetích stran (např. Cortex-M1, Vexriscv atd.). Výhodou je velká flexibilita a jejich snadnější implementace, ale za cenu nižšího výkonu periférie.

Implementace vlastních periférií v jazycích popisující hardware může být komplexní úloha vyžadující hodně zdrojů a financí. Můžeme ale využít již existujících, volně přístupných implementací (pokud existují) nebo některé řešení zakoupit.

Ukázkou specializovaného systému s možností jeho úpravy pomocí programu je periférie PIO (programovatelné I/O) uvnitř mikrokontroléru RP2040 [17]. Tato periférie je navržená pro implementaci rozhraní vestavěných systémů pomocí krátkého skriptu. Vývojář tak může vytvořit novou periferii na úrovni hardware, která by jinak musela být emulována pomocí software nebo pokryta externí součástkou. Lze tak například vytvořit periférii pro řízení VGA nebo HDMI signálu nebo proprietární rozhraní. Periférie se skládá ze 4 automatů a malé paměti. Paměť obsahuje 32 instrukcí a je společná pro všechny automaty. Každý automat má přístup k pinům mikrokontroléru, vstupní a výstupní frontě a k přerušení pro procesor. Zároveň má každý automat k dispozici aktuálně vykonávanou instrukci a dva pracovní registry.

Automat pracuje s 9 instrukcemi. Instrukční sada je navržena speciálně tak, aby každá instrukce zabrala jeden krok výpočtu a zároveň umožňovala v každém kroku automatu



Obrázek 2.4: Architektura periférie PIO [17].

ovlivnit GPIO výstup. Vzhledem k velikosti instrukční paměti je zde také kladen silný důraz na co největší kompaktnost výsledného programu. I takto jednoduchá instrukční sada je však výpočetně úplná a dokáže emulovat širokou řadu rozhraní (UART, I2C, SPI atd.).

2.5 Testování vestavěných systémů

Testování je proces hledání chyb v systému. Ačkoliv se může zdát, že daleko lepší by bylo chybám předcházet, než je hledat. Realita je taková, že zcela bezchybný systém neexistuje. [9]

Infrastruktura

K testování nestačí mít pouze testovaný systém, ale potřebujeme zároveň i pomocnou infrastrukturu. Tu můžeme rozdělit do tří hlavních částí. Části potřebné pro provedení testu (testovací prostředí), vybavení podporující efektivní provádění testů (pomocné nástroje a automatizace) a zázemí pro zaměstnance. [9]

Testovací prostředí

Tři nejzákladnější prvky testovacího prostředí jsou:

- **Hardware/software/síť**. Testovaný systém může mít v průběhu vývoje různé podoby. Pro každou fázi vývoje tak musíme mít rozdílné testovací prostředí.
- **Databáze testů**. Hlavní moto testů je, aby byly opakovatelné. Musíme tedy někde ukládat potřebná testovací data.
- **Simulační a měřící nástroje**. Pokud potřebujeme testovat systém, který ještě není v provozuschopném stavu, musíme chybějící části simulovat. Systém má také výstupy, které potřebujeme měřit a analyzovat. [9]

Pomocné nástroje a automatizace

Pomocné nástroje pro testovaní nejsou nutností, ale často značně ulehčí proces testování. Diverzita testovacích nástrojů je obrovská. Můžeme je ale klasifikovat podle aktivit, při kterých je primárně používáme.

- Plánování a řízení – konfigurace prostředí, plánování postupu testování, řešení chyb.
- Příprava – správa požadavků, analýza složitosti.
- Specifikace – generátor testovacích dat, generátor testovaných případů.
- Vykonání – exekuce testů, monitorování, porovnávání výsledků, analýza pokrytí, logování. [9]

Typy testů

Systém může být testovaný z mnoha úhlů – funkcionalita, výkon, spolehlivost atd. Při provádění testu obvykle ověřujeme několik těchto kvalitativních atributů najednou. To vede ke klasifikaci testů podle typu. Typ testu je tedy skupina aktivit s cílem evaluovat systém na množině kvalitativních atributů. Typ testu určuje, co bude testováno a co ne. Např. při testování funkcionality displeje nás nezajímá výše obnovovací frekvence, ale jestli displej zobrazuje to, co požadujeme. [9]

Úroveň testů

Úroveň testů je skupina činností, která je organizována a řízena jako celek a určuje, kdo test provede a kdy ho provede. Samotná úroveň se pak odvíjí podle aktuální fáze vývoje. Rozdělení testů do úrovní napomáhá k lepší struktuře testovacího procesu a hlavně umožňuje vytvářet testy v průběhu celého vývoje. Můžeme tak provádět testy na malých izolovaných komponentách, které, až dosáhnou požadované kvality, integrujeme do větších komponent a ty znovu testujeme oproti požadavkům. Vytváří se nám tak spektrum testů, od malých testů, které testují nejmenší komponenty v izolaci (unit testy), až po velké testy, které testují systém jako celek v reálném prostředí a s minimální mírou simulace. [9]

Automatické testování

Komplexita vestavěných systému se zvětšuje a s tím i komplexita testovacího prostředí. Zvyšuje se tak i snaha automatizace testování, aby vývojář nemusel trávit velké množství času s nastavením testovacího prostředí a spouštěním testů.

Při automatickém testování nejdříve řešíme testovací prostředí a jeho přizpůsobení na automatizaci. Testovací prostředí musí vyžadovat minimální, nejlépe žádnou, lidskou intervenci pro uskutečnění testů. Když je testovací prostředí připraveno, testovací systém sesbírá všechny testovací scénáře a připraví plán jejich exekuce. Po exekuci testů je nutné sesbírat výsledky testování a tyto výsledky prezentovat uživateli.

Využívání verzovacích systémů pro vývoj, jako je svn nebo git, je v dnešní době velmi populární. Automatické testování funguje velmi dobře ve spojení s těmito systémy. Pokud vývojář provede úpravu kódu systému, verzovací systém může tuto změnu detekovat a notifikovat testovací systém. Ten následně provede sadu testů pokrývající provedené změny a vyhodnotí jejich výsledek. Díky tomuto mechanismu tak docílíme toho, že alespoň na jednom místě je vždy funkční a správná verze systému.

Kapitola 3

Rozhraní

Základní funkcí vestavěného systému je snímání prostředí a jeho ovlivňování. Řídicí prvek typicky nepokrývá tuto funkci sám, ale deleguje ji na externí zdroje, např. měření zrychlení přenechá akcelerometru. Pro řízení celého systému jsou tak zapotřebí rozhraní zajišťující komunikaci mezi jednotlivými částmi systému. Pro různé případy využíváme různá rozhraní, jelikož každé rozhraní má své výhody a nevýhody. V rámci této sekce je popsáno dělení rozhraní z pohledu vzdálenosti od kontrolního prvku a zároveň jsou vybraná rozhraní blíže popsána. Toto dělení rozhraní nám ukáže odlišné motivace pro jejich použití a lépe reflektuje jejich použití v této práci.

Nejzákladnějším digitálním rozhraním je signál o hodnotě 0 nebo 1. Kombinací více signálů nebo přidáním času pak tvoříme komplexnější rozhraní. Pokud máme rozhraní, jehož součástí je hodinový signál, který zaručuje synchronizaci všech částí rozhraní, označujeme rozhraní jako synchronní. V opačném případě se jedná o rozhraní asynchronní. Jednotku informace můžeme brát jako jeden bit, častěji ale využíváme násobky osmi (8, 16, 32, 64 bitů). Pro vícebitové hodnoty nám nemusí stačit počet signálů rozhraní a je potřeba jednotku informace rozdělit na více částí — serializovat. Rozhraní, která dělí přenášenou informaci, označujeme jako sériová rozhraní. Ta, která v jeden čas dokáží přenést celou jednotku informace, nazýváme paralelní rozhraní. Použitím sériových rozhraní snižujeme propustnost systému, ale tím i jeho cenu. V komunikaci mezi zařízeními může být někdo, kdo ji řídí — master a druhý, který reaguje — slave.

3.1 Rozhraní uvnitř rozhodovacího prvku

Základem mikrokontroléru je jeho jádro, které řídí tok dat. To však není samo o sobě až tak užitečné a potřebuje ve své blízkosti další bloky (např. cache, ALU atd.). Interní rozhraní takového rozhodovacího prvku typicky komunikují na vysokých frekvencích a obsahují velká množství signálů. Vysoké frekvence vyžadujeme pro co nejvyšší výkon procesoru a větší počet signálů je, díky technologii výroby, ekonomicky udržitelný. Jako zástupce těchto rozhraní bylo vybráno rozhraní specifikace AXI, které je využíváno uvnitř FPGA řady Zynq od společnosti Xilinx.

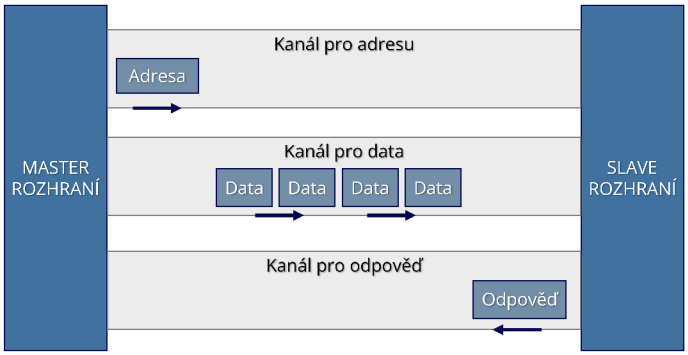
Rozhraní AXI4

AXI4 je vysokorychlostní master-slave rozhraní s nízkou latencí a možností dávkových přenosů dat mezi dvěma systémy. Pro komunikaci mezi více systémy je navzájem propojíme pomocí AXI interconnect. Rozhraní se skládá z pěti nezávislých kanálů:

- adresa ke čtení,
- přečtená data,
- adresa k zápisu,
- zapisovaná data,
- potvrzení zápisu.

Adresové kanály nesou informaci popisující přenášená data, samotná data putují po samostatném kanálu a každý zápis je potvrzen slave zařízením. Jak kanály pro zápis, tak i ty pro čtení, podporují dávkovou operaci. Každá dávka začíná specifikací adresy a kontrolních signálů, poté následuje až 256 balíku dat. [16]

Pro text této práce není nutné znát implementační detaily tohoto rozhraní, stačí znát pouze představené hlavní myšlenky rozhraní.



Obrázek 3.1: Transakce dávkového zápisu na rozhraní AXI4. [16]

Rozhraní AXI4-Lite

Jak název rozhraní napovídá, AXI4-Lite je odlehčená verze AXI4 rozhraní. Hlavní rozdíly spočívají v tom, že lze posílat transakce pouze o délce 1 a lze přistupovat pouze na celá data. [16]

Rozhraní AXI4-Stream

Je master-slave rozhraní pro přenos libovolně velkých dat mezi dvěma systémy, bez nutnosti adresování. Základem rozhraní jsou tři signály *TVALID*, *TREADY* a *TDATA*. *TVALID* signalizuje validní data ze strany mastera. *TREADY* signalizuje, že slave může přijmout data a *TDATA* obsahuje přenášená data. Data jsou přenesena pouze tehdy když *TVALID* a *TREADY* jsou aktivní. Tento základ pak může být rozšířen o další signály, které usnadňují směrování dat nebo jejich shlukování do paketů. [15]

3.2 Rozhraní vůči desce plošných spojů

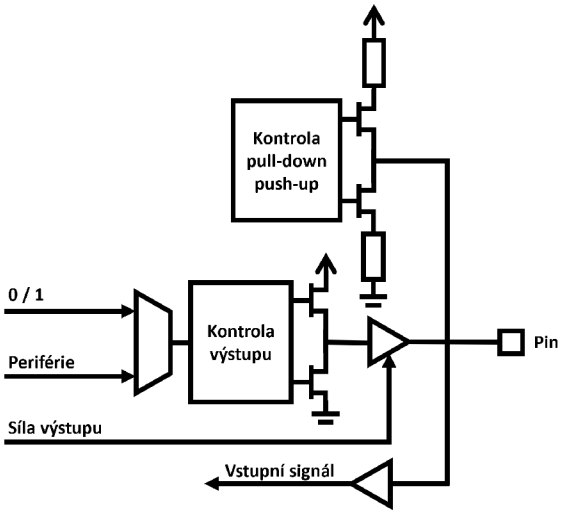
Samotný rozhodovací prvek nám k realizaci vestavěného systému obvykle nestačí. Pokud chceme snímat zrychlení, potřebujeme akcelerometr, případně, pokud potřebujeme více paměti, použijeme externí paměť. Se všemi těmito zařízeními potřebuje rozhodovací prvek

komunikovat, předávat data. Využívaná rozhraní pro tuto komunikaci obvykle obsahují oproti interním rozhraním výrazně méně signálů o nižších frekvencích. To je způsobeno tím, že každý signál navíc vyžaduje vývod na pouzdru rozhodovacího prvku a zároveň místo na desce plošných spojů (DPS).

GPIO

Jediným propojením mezi čipem a externími součástkami jsou vývody. Jeden vývod pro jednu funkcionalitu by byl velmi neefektivní. Jelikož plocha čipů je omezená, existuje mechanismus, který nám umožní na jeden vývod přivést více funkcionalit. Plocha takového mechanismu je mnohonásobně menší než jeden vývod pouzdra. GPIO, neboli univerzální vstup/výstup (General Purpose Input Output), nám přesně toto umožňuje.

Pomocí GPIO volíme, jestli se daný vývod chová jako vstup (stav vysoké impedance) nebo výstup. Pro vstup můžeme mít možnost připojit pull-down a pull-up rezistor. Ty použijeme, pokud chceme docílit toho, že logická úroveň není natvrdo vynucena. Tedy že jiné zařízení může stav log. úrovně změnit bez toho, aniž by vznikl zkrat. To umožníme tak, že připojíme signál k logické úrovni přes rezistor. Pokud ho takto zapojíme k log. 1 jedná se o pull-up rezistor a pokud k log. 0 tak o pull-down rezistor. Tuto vlastnost využívá např. rozhraní I^2C. Pro výstup můžeme specifikovat jeho sílu (maximální proud) nebo také úroveň napětí. GPIO nemusí k vývodu připojit jenom logickou úroveň, ale může také připojit některou z interních periférií (obvykle označované jako alternativní funkce). Způsob připojení těchto periférií se liší výrobce od výrobce. Velmi často má každý pin omezenou množinu možných periférií. Existují ale také čipy s vnitřní maticí, která umožní připojení libovolné funkcionality na libovolný pin (např. čip nRF52832). Volnost mapování funkcionalit umožní jednodušší návrh DPS a schopnost platformy reagovat na změny vývoje.

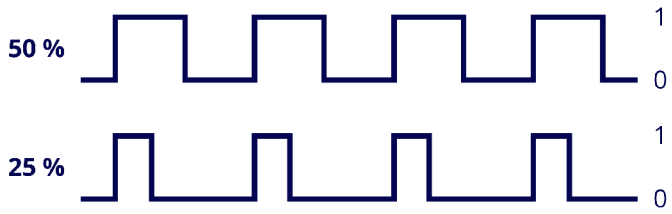


Obrázek 3.2: Ukázka možného zapojení GPIO.

PWM

PWM neboli pulsně šířková modulace je diskrétní metoda pro přenos spojitého signálu. Převod signálu spočívá v mapování spojité veličiny na střídu digitálního signálu. Střída značí

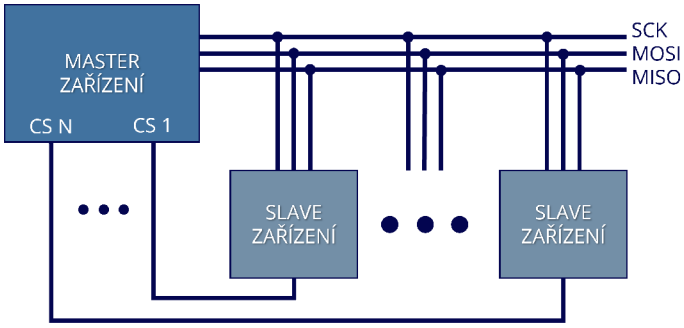
poměr logické 1 vůči logické 0 v jedné periodě. Díky možnosti mapovat na spojitou veličinu je PWM využíváno jako výstup senzorů (např. senzor otáček), jednoduchý DAC převodník, vstup aktuátorů (pozice serva, regulace jasu LED diody) atd. PWM ve vestavěných systémech obvykle tvoříme za pomoci časovače.



Obrázek 3.3: PWM signál na se střídou 50 % a 25 %.

SPI

SPI je duplexní sériové rozhraní. Rozhraní umožňuje připojení pouze jednoho master zařízení a N zařízení slave. Rozhraní obsahuje sběrnici o jednom hodinovém signálu (SCK), dvou datových linkách a N selektorů aktivního slave zařízení (CS n). Počet slave zařízení je omezen pouze počtem dostupných pinů. Na sběrnici může najednou probíhat komunikace pouze mezi dvěma zařízeními — a to nutně master a slave. Každá datová linka je tak vyhrazena pro jeden směr, od master do slave (MOSI) a od slave do mater (MISO).



Obrázek 3.4: Topologie rozhraní SPI.

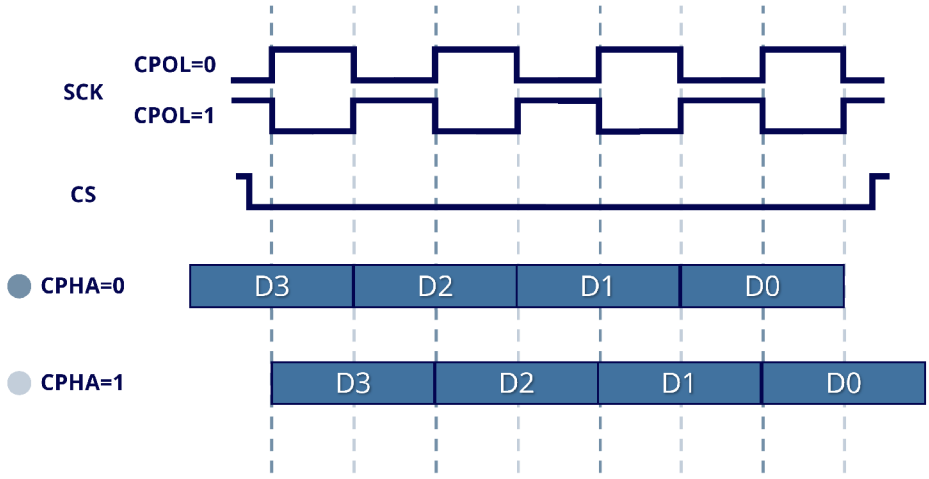
Průběh komunikace je následující:

1. Master nastaví CS do aktivní hodnoty pro zařízení, se kterým chce komunikovat.
2. Po definovaném zpoždění začne master generovat hodinový signál. Při čtecí hraně SCK čte master hodnotu MISO a slave MOSI. Při zapisovací hraně SCK master zapisuje přenášenou hodnotu na MOSI a slave na MISO. Komunikace pokračuje, dokud master generuje SCK a dokud je CS v aktivní úrovni.
3. Master zastaví generaci hodin, pokud byla přenesena všechna data a po definovaném zpoždění nastaví CS do neaktivní hodnoty.

SPI má čtyři různé módy komunikace. Mód rozhraní určuje polarita hodin (CPOL) a fáze hodin (CPHA). Polarita hodin určuje aktivní hodnotu hodin CPOL=0 je SCK v log. 0

a CPOL=1 je SCK v log. 1. Fáze hodin určuje čtecí a zapisovací hranu. Pro CPHA=0 je čtecí hrana při změně z neaktivní do aktivní hodnoty hodin a zapisovací hrana je při změně z aktivního hodnoty na neaktivní hodnotu hodin. Pro CPHA=1 je to přesně naopak. Detekce použitého módu není součástí protokolu a obě strany zařízení ho musí znát dopředu.

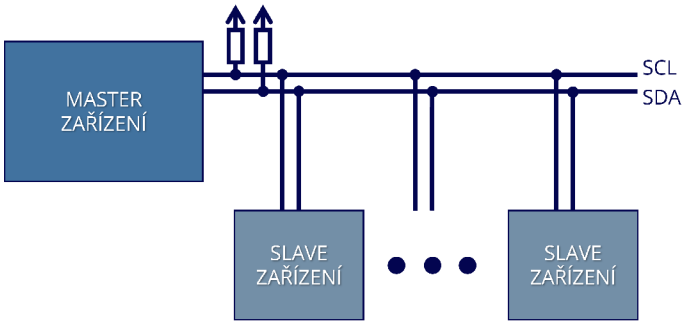
Existují i další varianty SPI jako jsou Dual SPI, Quad SPI atd. Ty rozšiřují počet datových vodičů a umožňují tak přenos více dat najednou.



Obrázek 3.5: Ukázka komunikace při různé konfiguraci CPOL a CPHA.

I²C

Sběrnice I²C (Inter-Integrated Circuit) je velmi levné, přesto efektivní rozhraní, nejčastěji využívané pro propojení s periferiemi. Sběrnice je half-duplex a komunikace je master-slave. Oproti SPI umožňuje také připojení více master zařízení na jednu sběrnici, má ale omezený počet slave zařízení kvůli šířce adresy. Sběrnice má jeden hodinový signál (SCL) a jeden datový pin (SDA). Typicky se sběrnice provozuje v předem dané frekvenci hodinového signálu, a to 100 KHz (standardní mód), 400 KHz (rychlý mód), 1 MHz (rychlý mód plus), 3,4 MHz (velmi rychlý mód). Každý vodič sběrnice je napevno připojen k log. úrovni 1 přes rezistor a zařízení ovlivňují logickou úroveň stáhnutím vodiče k zemi. Datový vodič tak může využívat několik zařízení v různých směrech bez rizika vzniku zkratu. [11]



Obrázek 3.6: Topologie rozhraní I²C.

Průběh komunikace na sběrnici je následující:

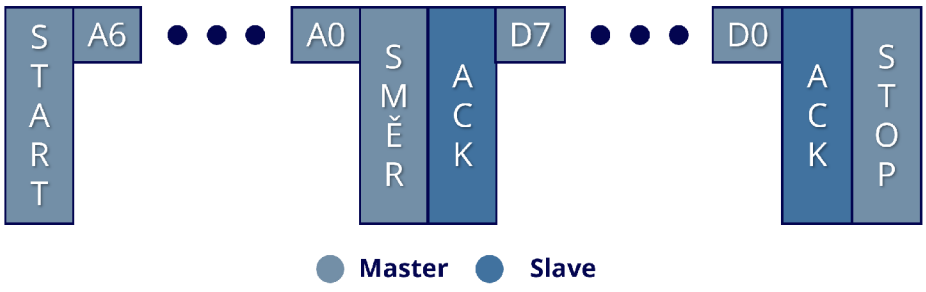
1. Sběrnice je v klidovém stavu SDA=1 a SDL=1.
2. Transakce začíná klesající hranou SDA.
3. Po definované hodnotě začíná master generovat hodinový signál. První případ, kdy je SCL=0, označujeme jako "start bit".
4. Přenášený bit je zapsán na SDA a jeho hodnota je přečtena při náběžné hraně SDL. Hodnota SDA se může změnit pouze při SDL = 0. Pokud by slave během komunikace nemohl další data přijmout, může nastavit SCL=0, což přinutí mastera čekat na uvolnění (čeká na SCL=1).
5. Po osmi přenesených bitech musí být přenos potvrzen. Master uvolní SDA a slave musí po dobu 1 bitu nastavit SDA=0. V opačném případě nastává chyba komunikace.

I^2C rozhraní může obsahovat více master zařízení, může tedy nastat situace, kdy dvě či více master zařízení začnou komunikaci ve stejný okamžik. Všechna master zařízení začnou komunikovat dle popisu výše. V jistou chvíli ale nastane mezi master zařízeními neshoda, někdo nastaví SDA=0, jiní očekávají SDA=1. Ti, co očekávali SDA=1 detekují SDA=0 a transakci ukončí – nastává pro ně chyba komunikace. Pokud neshoda nenastane, komunikace proběhla v pořádku. [11]

Oproti SPI nemá I^2C konkrétní pin pro adresaci slave zařízení. Adresace zařízení tak probíhá pomocí protokolu. Každé zařízení má svoji adresu, ta má 7 bitů a je zvolena výrobcem zařízení. Obvykle může uživatel upravit pár bitů adresy, aby mohl použít více stejných zařízení na jedné sběrnici. Pro každou komunikaci tak master specifikuje adresu, komu jsou data určena, a navíc nastaví směr dat. Pokud mu toto nějaké slave zařízení potvrdí, posílá/přijímá data. Existují vyhrazené adresy:

- 0x00 – slouží jako broadcast,
- 0x01 – indikuje, že bude následovat dlouhá komunikace.

Kvůli bitové šířce adresy a vyhrazeným adresám je počet adresovatelných zařízení omezen na 126. Existují však také varianty s 8, 9 nebo 10 bitovým adresováním. Ty však nejsou tak časté. [11]



Obrázek 3.7: Rámec I^2C se 7 bitovou adresou. Ukázka ukazuje zápis dat D na adresu A.

Broadcast adresuje všechna zařízení najednou a až druhý byte určuje jaká slave zařízení transakci potvrdí. Pro hodnotu druhého bytu: [11]

- 0x06 – Slave zařízení by měla provést reset a odpovědět s jejich adresou,
- 0x04 – Slave zařízení odpoví se svou adresou,
- jakékoliv jiné hodnoty s nejméně významným bitem rovným 0 jsou ignorovány,
- nejméně významný bit je roven 1 – Master sdílí svou adresu (v horních 7 bitech) ostatním Master zařízením.

UART

UART (Universal Asynchronous Receiver Transmiter) je duplexní asynchronní sériové rozhraní. Sběrnice propojuje zpravidla dvě zařízení a skládá se ze dvou vodičů. Z pohledu prvního zařízení je jeden pro příjem (RX) a druhý pro vysílání (TX), z pohledu druhého je tomu přesně naopak.

Jelikož komunikace není synchronní vůči hodinovému signálu, je zapotřebí mechanismus, pomocí kterého rozeznáme, kde končí a kde začíná jeden bit komunikace. Pro UART se zvolí předem daná frekvence tzv. baudrate (počet baudů/bitů za vteřinu), která určí šířku jednoho bitu. Typický baudrate je 9600, 38400 a 115200, ale existuje mnoho dalších. Dále se zvolí počet bitů, počet stop bitů a parita.

Klidová hodnota vodičů je v log. 1. Zde nastává problém. Pokud bychom chtěli poslat hodnotu FFh, nemáme jak určit začátek a konec. Proto začátek každé komunikace vyžaduje start bit (stáhnutí hodnoty k zemi po dobu jednoho bitu). Po odeslání všech dat jednoho rámce signalizuje vysílač konec transakce pomocí stop bitu (nastavení do aktivní hodnoty).

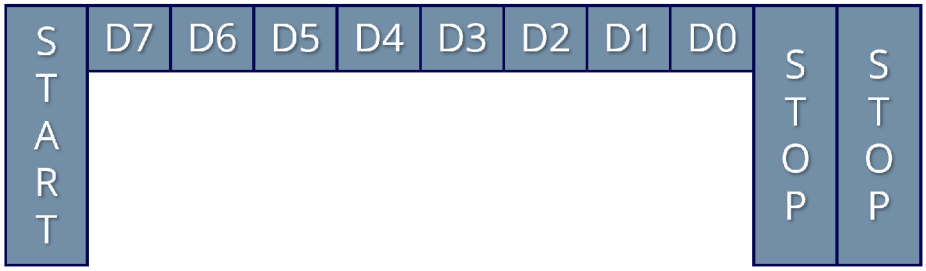
Pro detekci chyb v přenosu můžeme použít paritu. Pro UART se typicky používá parita sudá nebo lichá. Paritní bit by měl nabývat takové hodnoty, aby byl počet jedniček dle volby lichý nebo sudý.

Průběh odesílání dat je přímočarý a vysílač musí pouze dodržet dané časování a přidat režii. Průběh příjmu dat:

1. Přijímač detekuje začátek při klesající hraně signálu.
2. Podle nastavené frekvence začne načítat jednotlivé bity. Obvykle čteme data v půlce periody pro nejstabilnější výsledek.
3. Po načtení všech bitů přijímač ověří paritu (pokud je využita). Pokud je parita nesprávná, zahlásí chybu.
4. Čeká na daný počet stop bitů, jinak zahlásí chybu protokolu.
5. Pokud je signál v neaktivní hodnotě delší dobu, než je standardní doba pro jednu transakci, značí tato skutečnost přerušení komunikace a přijímač by měl vymazat přijímací buffer.

Analogové rozhraní

Digitální systémy mohou nativně pracovat pouze s digitálními signály. Pokud chceme pracovat se spojitými signály, potřebujeme převodník mezi těmito dvěma doménami. Využíváme tedy převodník analogového signálu na digitální (ADC) a převodník z digitálního signálu na analogový (DAC). Rozhodovací prvek často obsahuje jako jednu z dostupných periférií alespoň ADC, občas ale obsahuje i DAC. Zvládá tak s jistou mírou nepřesností zpracovat i analogové signály bez nutnosti externích převodníků.



Obrázek 3.8: Ukázka UART rámce pro 8 bitová data a 2 stop bity.

3.3 Rozhraní vůči okolí

Na této úrovni se krátce podíváme na dostupná rozhraní DPS, obsahující nějaký rozhodovací prvek. Tato rozhraní již nemusí být na napěťové úrovni kompatibilní s daným rozhodovacím prvkem a každý signál navíc znamená ještě větší finanční zátěž než u předchozích úrovní. Vzdálenost mezi dvěma zařízeními se také výrazně zvětšuje a oproti předchozím úrovním, kde se bavíme o vzdálenostech maximálně desítek centimetrů, je nyní možné komunikovat na vzdálenosti tisíce kilometrů a více. Vlivem okolí se nám začnou více projevovat parazitní efekty a jsou potřeba mechanismy, které ochrání integritu přenášených dat.

Dvě velmi rozdílné kategorie zařízení na této úrovni jsou zařízení drátová a bezdrátová. Mezi bezdrátové rozhraní můžeme zařadit např. Bluetooth, WiFi atd. Mezi drátové např. rozhraní CAN, hojně využívané v automobilovém průmyslu nebo RS485.

Kapitola 4

Popis problému

Tato práce si dává za cíl navrhnout platformu pro vývojáře a testery vestavěných systémů, která jim umožní vytvořit snadno konfigurovatelné, ovlivnitelné a pozorovatelné prostředí vestavěného systému. Hlavním cílem platformy je urychlit a zkvalitnit vývoj vestavěných systémů.

Vývoj vestavěného systému probíhá ve fázích, jak je popsáno v sekci 2.2. Tento nástroj je zaměřen na akceleraci v těch fázích, které vyžadují interakci s hardware. V počátcích vývoje slouží jako platforma pro rychlé prototypování a pro ověření konceptu. V průběhu vývoje se jeho použití mění na nástroj pro testování a podporu při ladění. V konečných fázích vývoje pak slouží jako dobrý základ pro automatické testování.

Na počátcích vývoje známe přibližnou podobu specifikace a představu o tom, jak bude finální systém zhruba vypadat. Začínáme tak rozmýšlet, jak daný systém implementovat. V tomto kroku pomůže navrhovaná platforma vývojáři rychle a snadno evaluovat kandidátní platformy. Po zvolení platformy nastává období, kdy hardware oddělení pracuje na prvním prototypu DPS. V tuto dobu má vývojář k dispozici pouze evaluační kity a devkity, které obvykle nemají všechna zařízení nacházející se ve finálním systému. Vývojář tak v mezičase využívá simulační nástroje, emulátory a případně provizorně propojuje více evaluačních kitů mezi sebou. Vytvářením simulačního prostředí a experimentálních zapojení ztrácí vývojář čas, který by mohl věnovat vývoji systému. Může zde také nastat problém, pokud daný výrobce používá proprietární technologii nebo nenabízí emulátor. V takovém případě je velmi obtížné vytvořit simulační prostředí PIL a musíme se uchýlit pouze k SIL. Problémy mohou také nastat, pokud je zařízení v okolí vestavěného systému nedostupné nebo je nepraktické, aby bylo v blízkosti vývojáře. Vývojář je pak nucen vynaložit značné úsilí pro simulaci chybějících částí nebo čekat na první prototyp od hardware týmu. Právě tento problém chce platforma řešit tím, že nabídne vývojaři vytvoření prostředí proti fyzickému rozhodovacímu prvku. Vyhneme se tak nadbytečné aproximaci na straně nejdůležitějšího prvku vestavěného systému, rozhodovacího prvku, a zároveň díky rozhraní pin-pin podporujeme nejširší možnou škálu zařízení. Omezením je pouze rychlost signálu, počet pinů a napěťová úroveň. Podíváme-li se totiž na rozhodovací prvek, jeho nejabstraktnější rozhraní s okolním světem je vývod a napětí na něm. Můžeme tak emulaci oddělit na spojitou a diskrétní. Pro spojité signály umožníme generaci funkcí a vzorů, nad kterým lze dále tvořit další zařízení pracující se spojitým signálem (např. potenciometr, baterie atd.). Obdobně pro diskrétní signály vytvoříme digitální rozhraní (SPI, UART atd.) a na nich konkrétní zařízení (např. akcelerometr). Vývoj přímo na vybraném rozhodovacím prvku má tu výhodu, že nezasahuje do způsobu jeho vývoje a odstraní čas strávený přípravou emulačního prostředí rozhodovacího prvku. Vývojář tak vyvíjí ve známém prostředí proti reálnému hardware obsahující reálné

periférie (ty výrobce zpravidla k simulaci neposkytuje). Zároveň ušetří čas strávený přípravou simulačního prostředí a soustředí se pouze na popis jednotlivých komponent a jejich chování.

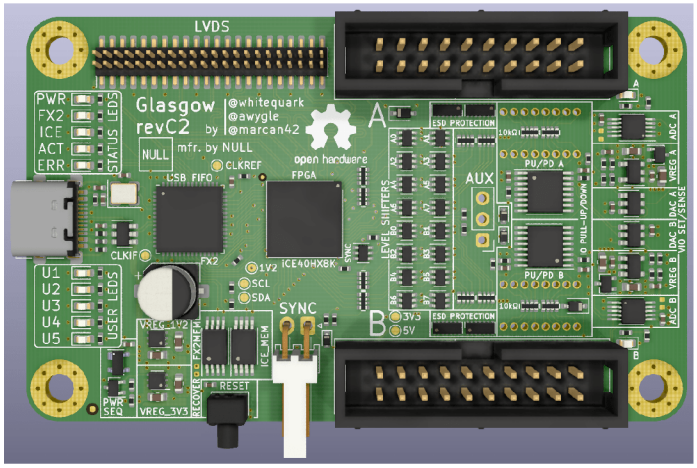
Po úvodní fázi vývoje, když jsou vyrobeny první prototypy systému, se simulované komponenty postupně nahrazují reálným hardware. Změní se rozhraní platformy a už nestačí komunikovat pouze s rozhodovacím prvkem. Nastanou tak potřeby generovat signály i o napětích mimo ty standardně používané ve sdělovací technice. Uvažme např. rozhodovací prvek, který je připojený k převodníku CAN a až výstup převodníku je výstupem testovaného systému. V této fázi jsou tak požadavky odlišné od těch původních. Tento problém se ale řeší i na úrovni samotného rozhodovacího prvku. U předchozího příkladu jsme potřebovali komunikovat se sběrnicí CAN a daný rozhodovací prvek tuto sběrnici nepodporoval, použili jsme tedy převodník. Stejný přístup můžeme použít i na druhé straně. Navrhovaná platforma umí komunikovat pouze se zařízeními na kompatibilní napěťové úrovni. Pokud tak uživateli umožníme vložit takovýto převodník do cesty mezi platformu a testovaný systém, jsme schopni komunikovat s ještě širším spektrem zařízení. Platforma se tak dokáže postupně přizpůsobovat potřebám vývoje už od počátku vývoje až k finálnímu produktu.

Vývojář vestavěných systémů během vývoje používá nemalé množství pomocných nástrojů. Mezi ty nejčastější patří multimetr, logický analyzátor, zdroj napětí a ladící sonda. Jelikož tyto zařízení pro jejich správnou funkci připojujeme k testovanému systému, je vhodné ty nejčastější nástroje integrovat uvnitř platformy. Motivací je minimalizovat počet potřebných zařízení pro běžné úkoly a vyžadovat specializované náčiní až pro komplexnější problémy. Integrované nástroje tak nemusí být příliš přesné, stačí pouze pokud poskytnou danou funkcionalitu v použitelné míře.

4.1 Existující řešení

Glasgow Debug Tool

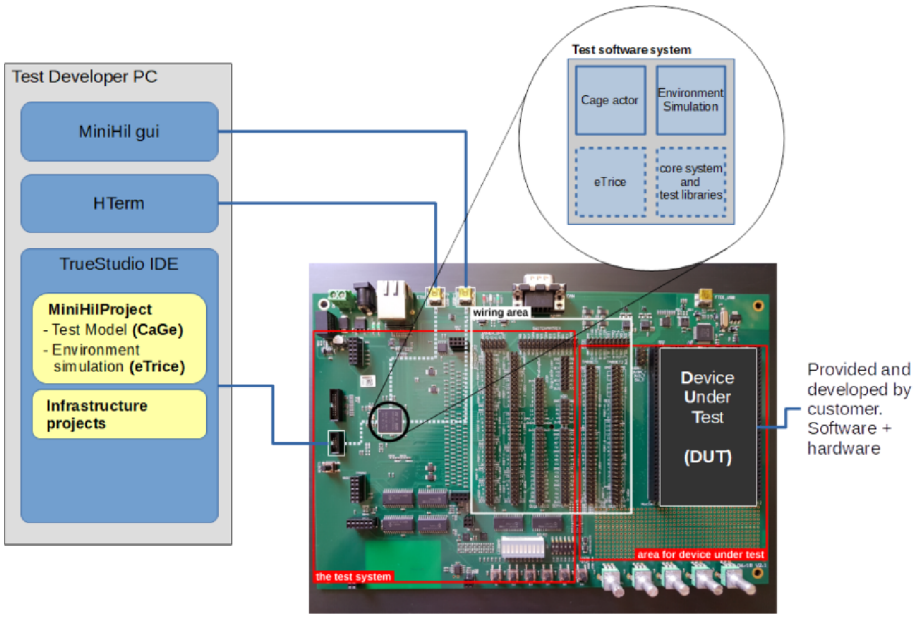
Glasgow je flexibilní nástroj pro průzkum digitálních rozhraní, který umožňuje uživateli vybrat z předpřipravených rozhraní a případně je i rozšířit o vlastní [1]. Nad vybranými rozhraními uživatel implementuje své případy užití. Platforma se skládá z podpůrné desky s FPGA, pomocných periférií a konzolové aplikace pro její obsluhu. Digitální rozhraní jsou implementována v FPGA a komunikují s PC pomocí USB FIFO rozhraní. Platforma se zaměřuje hlavně na digitální rozhraní, ale umožňuje také zpracování analogových signálů pomocí přídavných A/D a D/A převodníků. Celý projekt je postavený na open-source nástrojích, které jsou velmi dobře optimalizované. Glasgow si tak může dovolit sestavovat výsledný bitstream on-demand na straně uživatele. Platforma však pro snadné použití předpokládá, že využijete předpřipravených případů užití. Pro implementaci vlastního případu užití musí uživatel implementovat chování v jazyce Python a také definovat komunikaci s rozhraním v FPGA. Pro implementaci vlastního rozhraní musí mít uživatel znalosti návrhu číslicových obvodů a znalost knihovny Amaranth (knihovna pro popis RTL obvodů v jazyce Python). Hlavní myšlenka projektu je vytvořit nástroj pro snadné ladění rozhraní, nezabývá se přímo podporou vývoje.



Obrázek 4.1: Ukázka platformy Glasgow [1].

hitex miniHIL

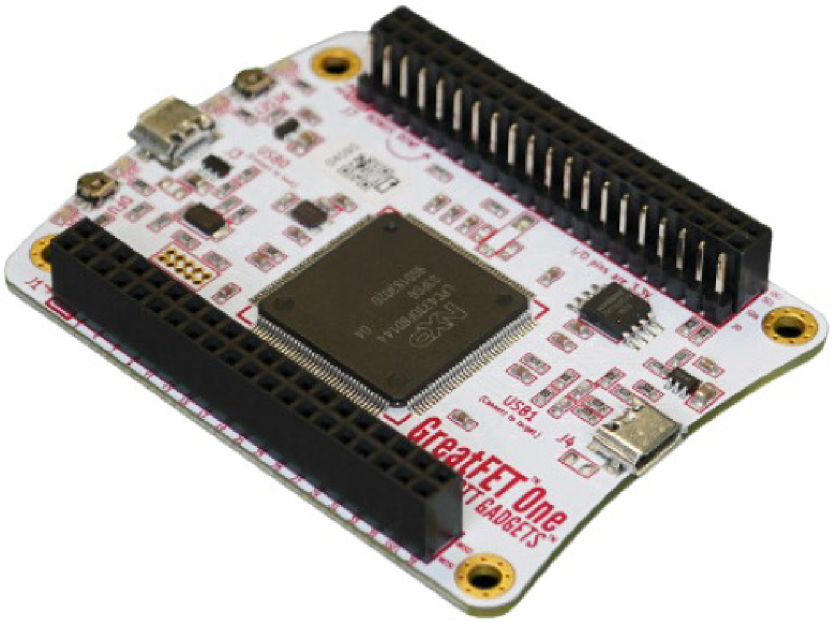
Řešení miniHIL od společnosti hitex je platforma s výkonným mikrokontrolérem, která simuluje okolí testovaného zařízení. Uživatel může simulovat běžně používaná digitální a analogová rozhraní a zařízení je využívající. Ovládání platformy je integrováno do vývojářské platformy TrueStudio IDE nebo pomocí konzolové či grafické aplikace. Motivací platformy je mít kompaktní HIL systém pro automatické testování, která zvýší testovací kvalitu a sníží cenu a čas vývoje produktu [4]. Celá platforma je však uzavřená. Existuje pouze málo informací a rozšíření platformy musí uživatel poptávat u výrobce.



Obrázek 4.2: Ukázka platformy miniHIL [4].

GreatFET One

GreatFET One je open-source platforma pro experimenty s digitálními rozhraními. Je postavena okolo výkonného mikrokontroléru, se kterým může uživatel komunikovat ze svého počítače. Uživatel si může vybrat z předpřipravených rozhraní, která vychází z periférií na daném mikrokontroléru. Ostatní rozhraní si může sám doimplementovat ve firmware. Platforma také zavádí systém "sousedů". Jedná se o standardizované rozhraní, přes které lze připojit k platformě další zařízení (např. převodníky, rádio atd.). Platformu tak lze rozšířit i o chybějící rozhraní.



Obrázek 4.3: Ukázka platformy GreatFet [3].

Vědecké články

Vědecké články zabývající se podobnou tématikou existují [27, 23]. Většinou se ale zabývají hlavně real-time emulací konkrétního zařízení a příliš neřeší použitelnost platformy z uživatelského hlediska.

4.2 Stanovení požadavků navrhované platformy

Pokud se podíváme na existující řešení, nejblíže navrhovanou platformu připomíná platforma miniHIL. Ta je však uzavřená a uživatelem nerozšířitelná. Ostatní podobné projekty nesdílí hlavní myšlenku navrhované platformy. Ačkoliv by bylo možné projekt Glasgow Debug Tool nebo GreatFET One použít jako základ a upravit jejich případ užití, byla by značně omezena použitelnost finálního zařízení. Horší platforma pro základ je GreatFET One jelikož obsahuje pouze mikrokontrolér. Množina použitelných hardware rozhraní by tak byla omezena a k dispozici by byla pouze ta, která jsou na mikrokontroléru. Ostatní by se musela implementovat v software nebo dodat externě. Lepší platforma pro základ je

tak Glagow Debug Tool. Obsahuje FPGA, rychlé propojení s PC a má podporu jak pro digitální, tak i analogové signály. Chybí jí však mikrokontrolér, na kterém by se mohlo implementovat chování zařízení. Implementace zařízení uvnitř FPGA přímo pomocí HDL byla zavrhnuta z důvodu komplexnosti vývoje a těžké rozšiřitelnosti uživatelem. Je zde také možnost implementovat v FPGA soft-procesor, to se však ukázalo jako nevhodné řešení z důvodu nedostatečných zdrojů použitého FPGA. Soft-procesor v FPGA nedosahuje tak velkých frekvencí (pod 150 MHz) a při využití pokročilejších funkcí zabírá příliš mnoho místa, které pak chybí pro implementaci platformy.

Vzhledem k výše uvedenému je mým cílem vytvořit vlastní platformu, která by byla kompaktní a cenově odpovídala ostatním laboratorním zařízením na trhu, tedy okolo 300 $. Tato platforma by měla mít FPGA pro implementaci konfigurovatelných rozhraní a vymezit tak výkon procesoru čistě pro simulaci a řízení. Dále by měla disponovat real-time operačním systémem (RTOS), který by umožnil deterministickou a opakovatelnou simulaci zařízení s nízkou latencí reakce. Platformu by měl být schopný rozšířit běžný vývojář vestavených systému a měla by mu umožnit pracovat jak s digitálními, tak i analogovými signály. Návrh platformy se soustředí na čtyři hlavní body: rozšiřitelnost, konfigurovatelnost, znovupoužitelnost a univerzalitu.

Rozšiřitelnost

Je výhodné, aby platforma řešila co možná nejvíce potřeb vývojáře. Není však možné pokrýt všechny. Je tak důležité, aby byl uživatel schopný platformu snadno rozšířit o vlastní řešení.

Konfigurovatelnost

Během testování potřebujeme simulované prostředí měnit a vytvořit tak co možná největší testovací scénář. Například při testování elektronického termostatu chceme simulovat různé komunikační rozhraní s topným systémem, různé teplotní křivky a kombinace uživatelských vstupů. Platforma tak musí umožňovat konfiguraci simulovaného prostředí.

Znovupoužitelnost

Použitím pinu mikrokontroléru jako rozhraní platformy zavedeme co možná nejobecnější rozhraní, na kterém stavíme všechny další elementy testovaného systému. Je však důležité zachovat znovupoužitelnost platformy, tedy její snadnou přenositelnost mezi různými vyvíjenými systémy. Umožníme tím dostat se rychleji k prvnímu funkčnímu konceptu a dokážeme se adaptovat na změny při jeho vývoji.

Univerzalita

Vývojář vestavěných systémů využívá při vývoji mnoho nástrojů (např. nástroje pro zprovoznění systému, hledání chyb a validaci jeho chování). Platforma se snaží identifikovat ty nejčastější a nejužitečnější nástroje, které vývojář využívá. Tyto nástroje následně integruje do jednoho celku tak, aby vývojáři nabídla univerzální platformu pro vývoj.

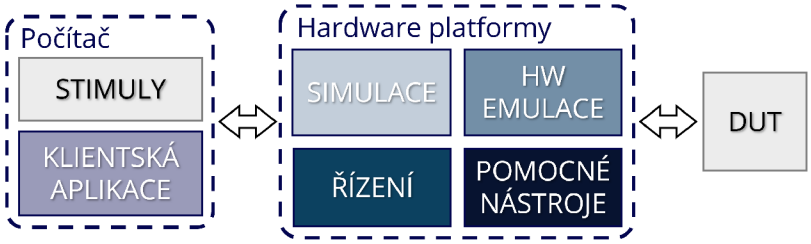
Kapitola 5

Návrh

V této kapitole je popsán celkový návrh platformy. Nejdříve je popsána celá platforma, z jakých hlavních částí se skládá a jaké jsou mezi nimi vazby. Následně je rozebrána struktura real-time simualce. Poté jsou detailněji popsány jednotlivé části platformy. Počínaje návrhem softwaru, který běží na uživatelském počítači. Dále návrh kódu běžící na fyzickém zařízení platformy. Až po popis návrhu uvnitř FPGA a návrhu specializované DPS pro integraci platformy do jednoho zařízení. Ve všech částech jsou probrány motivace návrhu s odůvodněním rozhodnutí, která se při návrhu provedla.

5.1 Platforma

V této sekci je detailněji popsáno, z jakých částí se platforma skládá a jaké byly motivace při jejím návrhu.



Obrázek 5.1: Systémový pohled na části platformy.

Celá platforma se skládá ze dvou částí. Klientské aplikace běžící na počítači uživatele a fyzického zařízení. Klientská aplikace je vstupním bodem pro uživatele, pomocí které může celou platformu konfigurovat a řídit. Fyzické zařízení je na jedné straně propojeno s počítačem a na druhé pak s testovaným zařízením (DUT). Úkolem fyzického zařízení je vytvořit real-time simulační prostředí a nabídnout vývojáři pomocné nástroje.

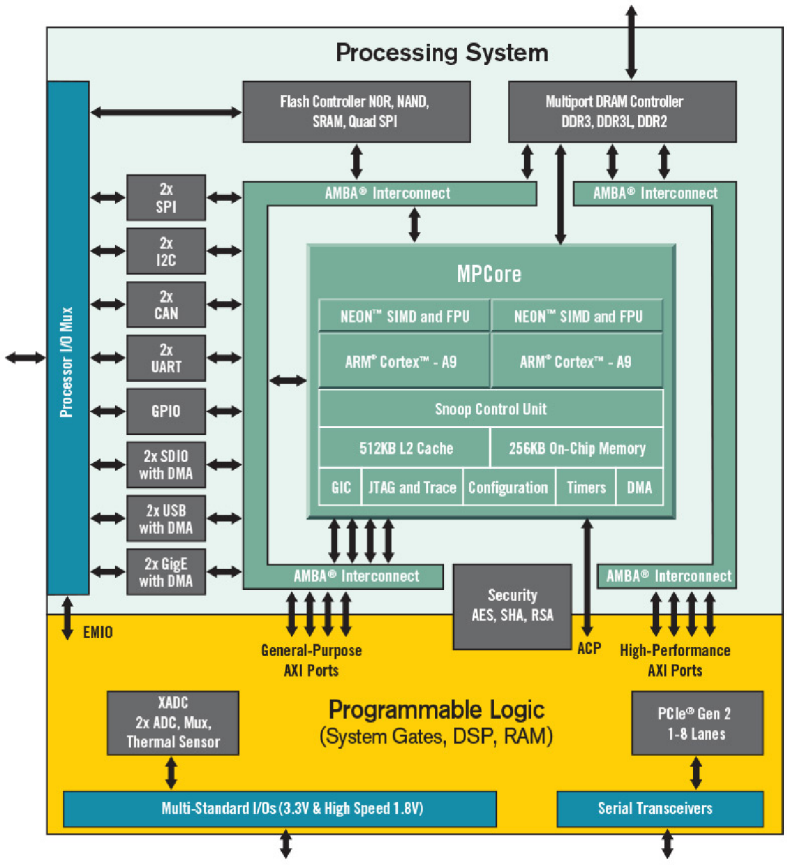
Výběr rozhodovacího prvku

Návrh platformy velmi ovlivňuje zvolený rozhodovací prvek, je proto vhodné ujasnit si tuto volbu na začátku návrhu. Požadavky na navrhovanou platformu značně limitují možný výběr. Při návrhu tak byly zváženy tyto možnosti:

- Samotné FPGA,

- FPGA a MCU,
- SoC (FPGA a MCU).

Použití samotného FPGA by vyžadovalo využití soft-procesorů uvnitř FPGA pro implementaci řídící logiky. To by s sebou přineslo celkově náročnější implementaci. Kromě implementace řídící logiky musíme najít vhodný procesor, správně ho nakonfigurovat, sestavit a vytvořit vývojové prostředí. Tento proces by byl o to komplikovanější, pokud bychom chtěli zajistit podporu operačního systému. Na základě toho bylo zvoleno použití dedikovaného procesoru. Pokud by bylo zvoleno MCU a FPGA jako dva čipy, muselo by se vytvořit vhodné rozhraní mezi MCU a FPGA. To není jednoduchý úkol, a navíc existují SoC řešení, která mají tento problém již vyřešený. Proto bylo zvoleno hotové řešení z dostupných vývojových kitů s SoC (FPGA a MCU). Jako nejvhodnější se ukázaly evaluační kity s SoC z rodiny Zynq 7000. Mezi ostatní uvažované patřily SoC z rodiny Zynq Ultrascale, které jsou pro tento návrh až příliš výkonné a hlavně drahé, nebo rodina Cyclone V. Tyto SoC jsou přímým konkurentem Zynq 7000 a mají téměř totožné parametry. Nicméně, SoC Zynq 7000 má dostupnou větší nabídku evaluačních kitů a lepší podporu vývoje.



Obrázek 5.2: Architektura rodiny Zynq 7000 [6].

Rodina SoC Zynq 7000 nabízí škálu SoC s dvoujádrovým aplikačním procesorem a FPGA. Konkrétní kusy se od sebe liší hlavně velikostí obsaženého FPGA a mírnými modifikacemi použitého CPU. Lze tak během vývoje snadno vyměnit použitý čip za slabší nebo silnější

podle potřeby. Komunikace mezi CPU a zbytkem SoC je řešena pomocí propojovací sítě AMBA Interconnect. Se samotným FPGA pak CPU komunikuje pomocí rozhraní AXI, které se z pohledu CPU tváří jako lokace v paměti. Kromě FPGA a CPU obsahuje SoC také periférie typické pro věstavěné systémy, DRAM kontrolér (pro až 1 GB RAM) a gigabitový ethernet. Ten bude sloužit jako vysokorychlostní propojení mezi platformou a počítačem.

Řídící systém

Pro návrh byl vybrán SoC rodiny Zynq 7000 a byly popsány motivace pro obsažení real-time prostředí. K dispozici je tedy dvoujádrový procesor a několik možností, jak splnit požadavek na real-time prostředí. Během návrhu byly zváženy tyto možnosti:

- RTOS běžící na obou jádrech,
- OS Linux na jednom jádru a RTOS na druhém,
- OS Linux s real-time rozšířením.

RTOS běžící na obou jádrech byl vyloučen jako první z důvodu, že zařízení bude komunikovat po síti a s dalšími zařízeními pomocí USB rozhraní. Správná a udržitelná implementace těchto rozhraní není triviální a v rámci této práce by byla příliš detailní. OS Linux řeší tyto problémy za nás a představuje odladěná rozhraní, se kterými můžeme ze systému interagovat. Další motivací pro OS Linux je velká míra dostupných hotových řešení a podpora skriptovacích jazyků. Za účelem urychlení vývoje byl tak, i za cenu vyšších požadavků na zdroje, zvolen jako hlavní systém OS Linux.

Zajištění real-time funkcionality je možné dosáhnout i přímo v OS Linux. Existují rozšíření standardní verze, která upravují kernel systému a přidávají preemptivní chování. Nejčastější způsob jejich implementace je představení druhého kernelu, který má na starosti real-time úlohy a funguje jako vrstva mezi hardware a standardním Linux kernelem [22]. Nejznámější open-source varianty jsou RTLinux, Xenomai a RTAI. Existuje ale i patch Linux kernelu, který dodává preemptivní chování a zachovává pouze jeden kernel. Nevýhodou přístupu s více kernely je přidaná komplexita komunikace mezi nimi, na druhou stranu preemptivní patch se nechová vždy real-time [22]. Průzkum byl zaměřen převážně na rozšíření Xenomai, ale ostatní varianty jsou velmi podobné. Jeho použití nijak neovlivňuje vývoj obyčejných aplikací, jejich vývoj probíhá stejně jako bez rozšíření. V případě, že chceme vytvořit real-time úlohu, použijeme speciální API pro komunikaci s real-time kernelem. Při vytvoření úlohy můžeme také specifikovat její prioritu pro vykonání. Rozšíření definuje také API pro komunikaci mezi úlohami a kontrolu časovačů. Jinak ale máme přístup ke všem zdrojům, tak jako z normálního systémového procesu.

Druhým uvažovaným přístupem je asymetrický multi-procesing (AMP). Jedná se o systém o více procesorech, které nemusejí být nutně stejného typu nebo architektury. Každý procesor v takovém systému má svůj adresový prostor a na každém z nich může běžet různý operační systém. [7] Je tak například možné, aby na jednom procesoru běžel operační systém Linux a na druhém RTOS. Známý framework umožňující tento přístup, který je zároveň podporovaný pro SoC Zynq 7000, je OpenAMP. Jedná se o framework, který kombinací dalších knihoven vytváří abstrakční vrstvu nad hardware a nabízí jednotné API pro všechny podporované systémy. Zároveň zajišťuje podporu pro komunikaci mezi jednotlivými systémy. Podporované systémy jsou Linux, FreeRTOS a aplikace bez OS.

Výběr vhodného přístupu byl zaměřen na porovnání mezi rozšířením Xenomai a frameworkem OpenAMP. Porovnání v tabulce 5.1 ukazuje hlavní výhody a nevýhody těchto přístupů pro použití v navrhované platformě. Hlavní výhodou přístupu Xenomai je jeho

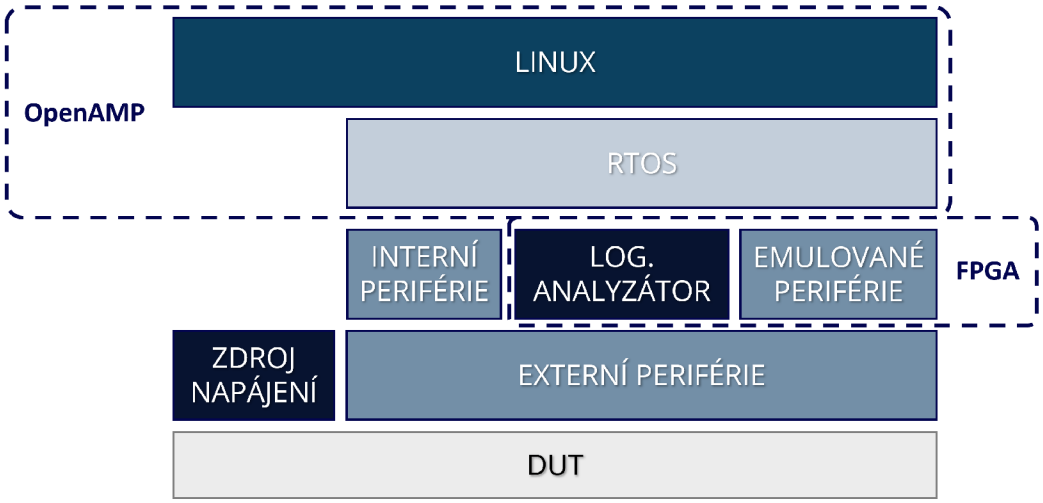
Tabulka 5.1: Provnání vhdonosti přístupů pro navrhovanou platformu.

Xenomai		**OpenAMP**	
Výhody	**Nevýhody**	**Výhody**	**Nevýhody**
Integrace s OS Linux a podobný vývojový cyklus.	Speciální API pro obsluhu RT sekce.	RTOS prostředí je známější vývojáři vestavěných systémů.	Staticky vyhrazená část paměti a procesor.
Real-time úlohy a normální úlohy sdílí stejný adresový prostor.	Strohá dokumentace.	Podporováno výrobcem.	Extra komunikace mezi systémy.
Podpora sdílených knihoven a možnost zkompilovat aplikaci jako osamocenou část.	Komplexnější sestavení a správné nastavení systému.	Dva oddělené systémy. Menší šance, že simulace shodí hlavní systém.	Nelze triviálně rozšiřovat bez rekompilace celého systému.
Umožňuje dynamickou utilizaci obou procesorů.			

integrace s OS Linux. V real-time simulaci máme dostupné oba procesory a celou paměť, zároveň máme přístup ke všem prostředkům, které OS Linux nabízí. Oproti tomu v OpenAMP máme dva nezávislé systémy, kde každému musíme staticky vyhradit část paměti a přiřadit jádro procesoru. U Xenomai můžeme brát jednotlivé úlohy jako samostatné programy a snadno dynamicky dodat nový kód. U OpenAMP se pouští celý systém najednou a FreeRTOS nepodporuje dodání nového chování dynamicky. Pro každé nové rozšíření tak musíme celý systém překompilovat a restartovat. Nicméně velkou výhodou přístupu OpenAMP je, že je podporován nástroji výrobce rozhodovacího prvku. FreeRTOS je také velmi známé prostředí ve světě vestavěných systémů a pro koncového uživatele, vývojáře vestavěných systémů, je tak snazší pro použití než API Xenomai. Pro finální návrh byl zvolen přístup OpenAMP, který je vhodnější pro koncového uživatele a umožní hladší průběh implementace díky podpoře výrobce. Nicméně, pokud by byl větší prostor pro průzkum správného sestavení Xenomai s OS Linux a upravení API, aby bylo vhodnější pro širší publikum, bylo by zvoleno rozšíření Xenomai.

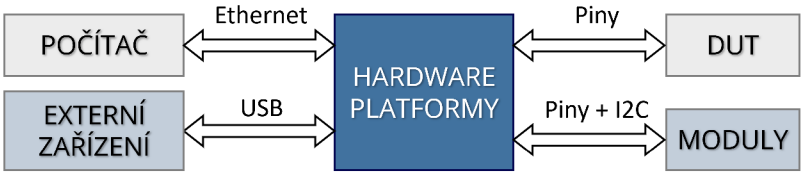
Architektura platformy

Podíváme-li se blíže na strukturu fyzické části platformy (obrázek 5.3), řízení celé platformy má na starosti operační systém Linux. Ten přímo ovládá zdroj napájení a real-time operační systém (RTOS). RTOS vytváří simulační prostředí, které využívá interní, emulované a externí periférie. Interní periférie značí periférie obsažené uvnitř SoC (např. I^2C), emulované periférie jsou periférie implementované uvnitř FPGA a externí periférie jsou pomocné periférie mimo SoC na DPS (např. DAC, generátor funkcí atd.). Dále RTOS také ovládá



Obrázek 5.3: Pohled na hlavní části fyzické platformy.

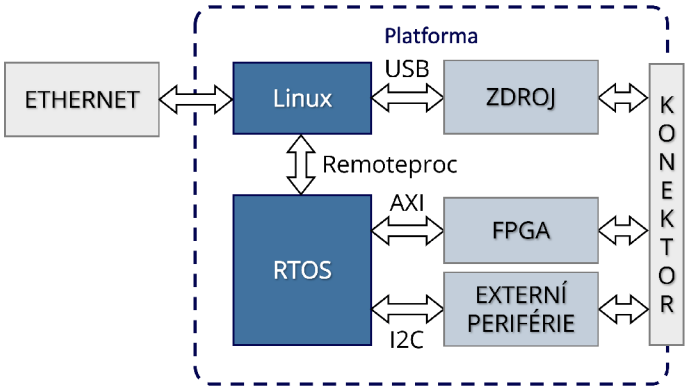
pomocná zařízení uvnitř FPGA (např. logický analyzátor). Testované zařízení pak využívá zdroje napájení a signálů simulovaného prostředí.



Obrázek 5.4: Komunikace platformy vůči okolí.

Pokud se podíváme na návrh platformy z pohledu komunikace mezi jejími částmi, můžeme návrh rozlišit na komunikaci uvnitř platformy a mimo ni. Pro vnější komunikaci počítá návrh se čtyřmi druhy zařízení. Uživatelským počítačem, testovaným systémem, externími zařízeními a rozšiřujícími moduly. Hlavní komunikační rozhraní platformy je ethernetové rozhraní a slouží pro připojení k počítači uživatele. Toto rozhraní slouží pro konfiguraci a řízení platformy dále také pro přenos simulovaných dat. Rozhraní je dostatečně výkonné (1 Gb/s), aby přenášelo například obrazová data simulovaného displeje. Přes USB rozhraní je k platformě připojený zdroj napětí. Toto rozhraní také umožňuje budoucí rozšíření o další zařízení, jako je např. programátor nebo ladící sonda. Umožníme tak platformu přizpůsobit pro lepší podporu automatického testování. Druhé velmi důležité rozhraní je rozhraní k samotnému testovanému systému. To musí být co možná nejvíce generické a umožnit připojení ideálně libovolného testovaného systému. Do vnějšího rozhraní jsou započítané také rozšiřující moduly. Ty jsou více rozebrány v sekci 5.6, stručně se ale jedná o způsob, jak může uživatel platformu rozšířit o signály atypické v prostředí rozhodovacího prvku.

OS Linux zpracovává příchozí komunikaci přes ethernet a přeposílá požadavky z klientské aplikace do RTOS nebo do zdroje napájení. Komunikace mezi OS Linux a RTOS probíhá přes knihovnu remoteproc, která je určena použitím frameworku OpenAMP. OS Linux je v této komunikaci v roli mastera. Velmi důležitá komunikační linka je také mezi RTOS a FPGA. Ta se z pohledu RTOS tváří jako lokace v paměti a z pohledu FPGA to je rozhraní AXI. Přes tuto linku probíhá veškeré řízení periférií uvnitř FPGA a přenos



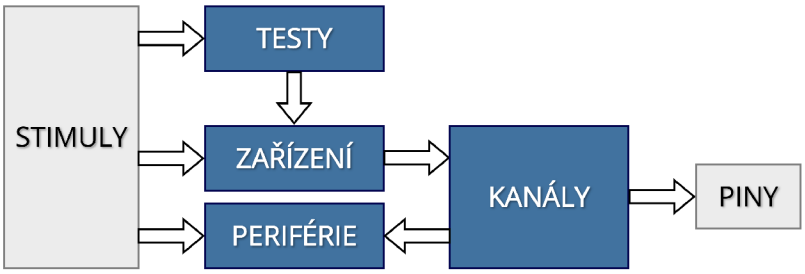
Obrázek 5.5: Komunikace uvnitř platformy.

přijatých/odesílaných dat. Zároveň se přes FPGA mohou propojit s výstupním konektorem i interní periférie použitého SoC. Pro volitelné externí periférie na DPS bylo zvoleno jako hlavní komunikační rozhraní I^2C. Externí periférie jsou na obrázku 5.5 připojeny k výstupnímu konektoru, více detailů k této vazbě je blíže vysvětleno v sekci 5.6. V poslední řadě je ke konektoru připojen i variabilní zdroj napětí, který dodává energii testovanému systému. Ten je konfigurován přes USB rozhraní z OS Linux.

5.2 Simulace prostředí testovaného systému

Obrázek 5.6: Entity vyskytující se v simulaci.

Jedním z hlavních cílů platformy je vytvořit simulační prostředí. Jeho návrh počítá se dvěma hlavními entitami. První z nich představuje libovolné zařízení, které by mohl chtít vývojář simulovat. Druhá pak periférii (emulovanou, externí nebo interní). Obě tyto entity mají atributy a akce, se kterými lze interagovat. Důvod pro zavedení akcí a atributů byla snaha vytvořit generické rozhraní, přes které lze simulované prostředí snadno ovlivňovat. Uvažme tedy například zařízení krokový motor. Takové zařízení má určité parametry, které by vývojář mohl chtít během simulace upravovat nebo vyčítat. Pro motor je to například aktuální rychlost otáček, příkon, počet kroků atd. Akce pak označuje to, co může zařízení provést. Tedy konkrétně pro motor například běž, zastav, simuluj poruchu atd. V navrhované simulaci musí mít každé zařízení přiřazenou nějakou periférii. Je na konkrétní implementaci zařízení, jestli danou periférií bude jen vstup/výstup nebo něco komplexnějšího. U periférie máme atributy a akce ze stejného důvodu jako u zařízení. Např. u rozhraní UART můžeme nastavit atributy baudrate nebo použitou paritu a akce může být např. simuluj chybu parity. Obecně tedy atributy mění charakteristiky simulace a akce provádí změny prostředí. Pomocí tohoto jednoduchého rozhraní jsme schopni bez omezení vytvářet dostatečně složité simulační scénáře.



Obrázek 5.7: Struktura objektů uvnitř simulačního prostředí.

Výsledná struktura simulace je ukázána na obrázku 5.7. Nejvyšším možným objektem v hierarchii simulace je test, který popisuje testovací scénář a k exekuci využívá vybraná zařízení. Tento objekt nemusí být použit vždy. Použijeme ho pouze v případě, pokud chceme vytvořit test v real-time prostředí. Pro obyčejné testy využijeme API klientské aplikace. Ostatní objekty je pak nutné použít vždy. Každá periférie obsahuje kanály, jedná se o označení pro jednotlivé vodiče konkrétní periférie (například periférie UART obsahuje 2 kanály — jeden pro příjem a druhý pro odesílání). Kanály slouží pro identifikaci vodičů rozhraní a jejich typu. Tento přístup nám umožní granulárnější kontrolu nad tím, jaké piny používá daná periférie. Zařízení si pak vybírá kanály konkrétních periférií. Je tak možné periférie sdílet napříč více zařízeními s tím omezením, že v danou chvíli může zapisovat do odesílacího kanálu pouze jedno zařízení, čtení je sdíleno mezi všemi připojenými zařízeními. Tato vlastnost je důležitá například pro simulované prostředí obsahující jednu periférii I^2C v režimu slave a dvě různá slave zařízení, která jsou k ní připojená. Každé zařízení tak potřebuje obdržet zprávu, zkontrolovat adresu a případně odpovědět. Pokud v simulaci použijeme interní periférii, navedeme její výstup do FPGA a dále s ním pracujeme jako s výstupem emulované periférie. Speciálním případem jsou periférie externí. Ty ke své funkci vyžadují minimálně jednu interní nebo emulovanou periférii. Výsledná periférie tak musí obsahovat jak chování využité periférie, tak i chování externí periférie.

5.3 Softwarové rozhraní navržené platformy

Tato sekce je zaměřena na návrh softwarové části platformy. Sekce se dělí na část běžící na počítači uživatele a na část běžící na fyzickém zařízení platformy v prostředí OS Linux.

Aplikace na počítači uživatele

Pro komunikaci s fyzickým zařízením platformy je potřeba navrhnout aplikaci, pomocí které bude mít uživatel možnost platformu ovládat a upravovat. Aktuální návrh počítá pouze s návrhem knihovny a konzolové aplikace, nicméně nechává prostor pro možné rozšíření o grafické rozhraní.

Knihovna implementuje operace pro připojení se k platformě a komunikaci s platformou. K implementaci komunikace využívá API fyzického zařízení a převádí ho do vhodné podoby pro zvolený programovací jazyk.

Konzolová aplikace je strukturována jako sada nástrojů, které jsou na sobě nezávislé a využívají funkce knihovny. Mezi uvažované nástroje patří:

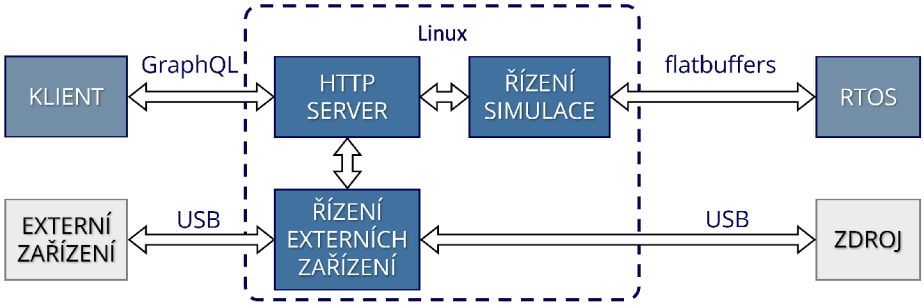
- Interaktivní mód, který připraví vývojáři prostředí pro experimentování.

- Aktualizace fyzického zařízení na novější verzi.
- Překlad skriptů pro periférii PIO.
- Překlad simulace.

Motivací pro vytvoření konzolové aplikace je tedy především nabídnout vývojáři sadu nástrojů pro správu platformy a experimenty.

Aplikace na fyzickém zařízení

Vzhledem k použitému rozhraní ke komunikaci s počítačem je aplikace koncipována jako HTTP server. Mimo obsluhy serveru má aplikace na starosti také řízení USB zařízení a komunikaci se simulačním prostředím v RTOS.



Obrázek 5.8: Struktura aplikace na fyzickém zařízením.

Nad HTTP komunikací byl vybrán ještě protokol GraphQL, který za nás řeší zpracování požadavků klienta. GraphQL je dotazovací jazyk pro vytváření API a prostředí pro exekuci těchto dotazů. GraphQL poskytuje úplný a srozumitelný popis dat a dává uživateli možnost dotazovat přesně ta data, která potřebuje. [2] Toto rozhraní bylo zvoleno, oproti tradičnějšímu přístupu přes REST API, jelikož snižuje potřebnou implementaci na straně fyzického zařízení a tím umožní rychlejší vývoj. Je to dosaženo tím, že server pouze definuje objekty a jejich atributy a je na druhé straně, aby vytvořila dotaz podle své potřeby. V případě, že uživatel chce provést změnu, GraphQL zavádí koncept mutací. Jedná se o seznam akcí podporovaných serverem, které určitým způsobem mění prostředí serveru. Tento přístup objektů s atributy a akcí, které mění prostředí, je velmi podobný navrženému přístupu u simulace. Tam jsou také objekty s atributy a podporovanými akcemi. Mělo by tak být přímočaré tyto dva přístupy propojit. Lze to vyzkoušet i na příkladu pro zařízení akcelerometr. Ten lze zjednodušeně definovat jako:

```
type Accelerometer {
    id: Int
    x: Float
    y: Float
    z: Float
}
```

Klientská strana při připojení k serveru dostane informace o všech dostupných objektech. Pokud se pak klient rozhodne, že ho zajímají pouze atributy `x` a `y`, může vytvořit svůj vlastní dotaz dotazující pouze tyto atributy:

```
{
    accelerometer(id: 0) { x, y }
}
```

V tom pouze uvede, jaký konkrétní akcelerometr požaduje a pak specifikuje všechny atributy, které ho zajímají. V případě, že bychom chtěli změnit stav akcelerometru, je situace trochu komplikovanější. Jelikož mutace není vázaná na konkrétní objekt a v principu je to jenom funkce. Pokud bychom tak chtěli třeba změnit atribut `x`, můžeme buď vytvořit konkrétní funkci pro daný objekt, nebo připravit generickou mutaci, kterou konkretizujeme pomocí argumentů. Příklad pro generickou mutaci by mohl vypadat takto:

```
setAttribute(type: "accelerometer", id: 0, attr: "x", value: 29.2) {
    ok
}
```

Generickou mutací sice zkomplikujeme změnu atributů, nicméně druhý způsob by nám velmi jednoduše zahltil jmenný prostor a zanesl by mnoho duplicitního kódu. Pro potřeby návrhu platformy je důležité, aby tyto definice objektů a mutací šly vytvářet dynamicky, protože uživatel mění prostředí simulace za běhu platformy. A to vybraný protokol umožňuje.

Důležitou součástí aplikace je samotné řízení simulace. Tato část se stará o spuštění simulace, kontrolu jejího stavu a o předávávání dat mezi počítačem a simulací. Stará se tak o nahrání binárního souboru do vyhrazené paměti, vymezení procesoru a předání řízení. Po inicializaci RTOS si zároveň zjistí, jaké zařízení a periférie systém obsahuje a jaké jsou jejich parametry. Tyto informace pak promítne do vnějšího API, dále pak přeposílá dotazy od klienta a kontroluje stav simulace. Komunikace mezi OS Linux a RTOS je zajištěna pomocí knihovny remoteproc poskytnuté frameworkem OpenAMP. Nicméně, tato knihovna posílá zprávu pouze jako binární data a je na uživateli, aby data správně formátoval. Proto je pro zjednodušení tvorby protokolu nad toto rozhraní přidána ještě knihovna flatbuffers. Knihovna flatbuffers je serializační knihovna, která umožňuje definovat protokol ve speciálním jazyce. Z této definice dokáže následně vygenerovat potřebné soubory pro některý z podporovaných programovcích jazyků. Lze tak snadno vytvořit komunikační kanál jak na straně OS Linux i na straně RTOS. Tato knihovna byla zvolena pro snadnou rozšiřitelnost a rychlou deserializaci bez nutnosti kopírování. Přes tento komunikační kanál tak putují všechny příkazy a dotazy od klienta a zároveň zprávy pro synchronizaci mezi řízením simulace a samotnou simulací.

Část pro obsluhu externích zařízení je navržena pouze pro rozhraní USB. Tato část je zamýšlena pro připojení dalších komplexních zařízení, která by vývojář mohl využít při testování. V tomto návrhu se uvažuje pouze s variabilním zdrojem napětí, pomocí kterého se bude testovaný systém napájet. Tato část programu tedy zajišťuje detekci připojených zařízení a vytvoření stejné struktury jako je tomu v simulačním prostředí. Z pohledu klienta tak s externím zařízením pracujeme totožně jako se simulovaným zařízením. Z pohledu simulace je možné zařízení také použít, musí ale proběhnout komunikace mezi OS Linux a doba vyhotovení požadavku není deterministická. Nicméně k rozhraní lze v budoucnu připojit i další zařízení, jako je např. programátor, ladící sonda nebo Wi-Fi adaptér. Pro jejich plnou podporu je však potřeba doimplementovat chování do této sekce. Rozšíření o další zařízení uživatelem tak v tomto návrhu není uvažováno. Je pouze zaveden univerzální způsob pro možná budoucí rozšíření.

5.4 Firmware RTOS jádra

Firmware sekce se zabývá návrhem kódu běžícím v prostředí RTOS, zaměřuje se tedy hlavně na vytvoření vhodného prostředí pro simulaci zařízení. Platforma nenabízí hotové prostředí simulace, ale dílnu s potřebnými nástroji, kde vývojář může pracovat. Tvorba simulačního prostředí je tedy hlavní část platformy, která je nejvíce ovlivněna vývojářem a je pod jeho plnou kontrolou.

Obrázek 5.9: Objekty simulačního prostředí v RTOS.

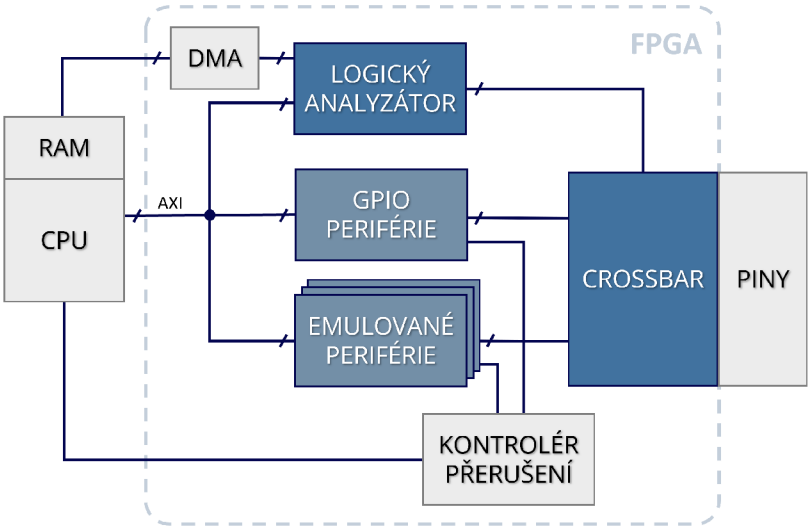
Objekty vyskytující se v simulaci jsou testy, zařízení a periférie. Periférie jsou implementovány uvnitř SoC, FPGA nebo mimo SoC a firmware musí obsahovat pomocný kód — driver, který umožní jejich řízení. Platforma musí obsahovat drivery pro všechny použité periférie. Při rozšíření platformy o novou periférii tak do této části musíme doplnit i její driver. Zařízení pak využívají jeden či více driverů a implementují nad nimi své chování. Hlavní tři částí zařízení, které musí vývojář uvést, jsou popis zařízení, inicializace zařízení a jeho chování. Popis zařízení obsahuje jeho jméno, atributy, akce a periférie, které využívá. Atributy se dále skládají ze jména, možnosti přístupu (čtení, zápis nebo čtení i zápis) a popisu, jak tento atribut získat nebo upravit. Akce mají opět jméno a popis, jak danou akci provést. Seznam periférií je potřebný, aby mohla simulační část zkontrolovat, že má všechny potřebné periférie před vytvořením zařízení. Inicializační část je exekuovaná pouze jednou při vytvoření zařízení a poté řízení přejde do části chování. To je nekonečná smyčka, která je periodicky volána. Zařízení běží v kontextu systémové úlohy RTOS. Každé zařízení má vyhrazenou jednu úlohu a vývojář může při vytvoření zařízení zvolit jeho prioritu vykonání vůči ostatním zařízením. Zároveň může využít primitiva pro komunikaci mezi úlohami, které RTOS nabízí. Test je pak pouze speciální instance zařízení, která kromě periférií uvádí ještě potřebná zařízení a má předem dané některé atributy.

Součástí firmware je také řízení simulace a komunikace s OS Linux. Tyto části zasahují do průběhu simulace, ale vývojář je ze simulace ovlivnit nemůže. Řízení simulace drží přehled o dostupných perifériích a udržuje informaci o tom, kdo vlastní jaký kanál periférie. Při vzniku nového zařízení ověří, jestli je mu možné přidělit potřebné kanály a pokračuje v inicializaci a ve vytvoření úlohy. Zároveň se stará o uvolnění nepotřebných zařízení. Komunikace s OS Linux zpracovává příchozí zprávy a odesílá je zpět.

5.5 Periférie v FPGA

FPGA uvnitř platformy slouží pro emulaci rozhraní, simulaci zařízení, směrování signálů z vnitřních periférií a pro implementaci specializovaných nástrojů. Je to tak pro uživa-

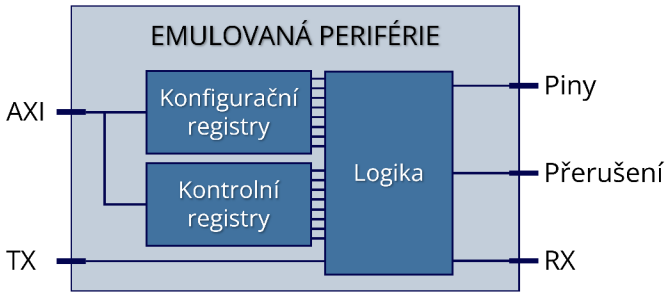
tele nejnižší možná úroveň, kterou může rozšiřovat a konfigurovat. Rozšiřitelnost na této úrovni je docílena možností přidat novou funkcionalitu, v rámci této práce označovanou jako emulovaná periférie. Uživatel tak může přidat své vlastní rozhraní nebo v hardwaru implementovat chování svého zařízení.



Obrázek 5.10: Architektura uvnitř FPGA.

Návrh struktury uvnitř FPGA je ukázán na obrázku 5.10. Hlavní komunikační rozhraní pro všechny funkční bloky uvnitř FPGA je sběrnice AXI. Ta je vytvořená a nativně podporována výrobcem FPGA a je vhodná pro snadné rozšíření o nová zařízení. Všechna zařízení připojená k této sběrnici pracují v režimu slave a jsou řízeny z CPU. Není tak potřeba další logika a všechny prvky na sběrnici pracují jako nezávislé celky. Kromě rozšiřitelné sekce je uvnitř FPGA vždy dostupná periférie GPIO. Ta v FPGA plní nejzákladnější úlohu — nastavení výstupu na logickou úroveň nebo vyčtení jeho stavu. Pokud tak nemáme vybranou žádnou periferii, stále je možné ovládat výstup a provádět základní úkony. Při experimentování nebo testování chce vývojář vidět stav signálů v konkrétním bodě, může tak například zjistit odezvu systému, odhalit chybu protokolu atd. Proto je v FPGA obsažen také logický analyzátor, který přesně tuto analýzu umožní. Výstupem logického analyzátoru je stav všech pinů v konkrétním bodě. Stav pinů se snímá při vybrané vzorkovací frekvenci, generuje tak velké množství dat. Z tohoto důvodu má vyhrazenou periférii DMA, která bere všechna vygenerovaná data a bez zátěže procesoru je vkládá přímo do paměti RAM. Jedním z požadavků platformy je připojení k evaluačním kitům. Mikrokontrolér na takovém kitu umožňuje vývojáři, díky GPIO, změnit, jaká periférie je na jakém pinu. Je tak v zájmu platformy umožnit tuto změnu reflektovat v simulačním prostředí, bez nutnosti něco fyzicky přepojovat. V FPGA je tak obsažen crossbar, který umožní periférii číst nebo zapisovat na libovolný pin.

Pro snadné rozšíření o nové periférie bylo potřeba zavést jednotné rozhraní pro všechny periférie. Toto rozhraní je znázorněno na obrázku 5.11. Návrh rozhraní předpokládá, že periférie má přes rozhraní AXI přístupné konfigurační a kontrolní registry, které ovlivňují logiku periférie. Předpokládá také, že logika dokáže ovlivnit pouze stav výstupních pinů a případně vyvolat přerušení pro procesor. Nepředpokládá se tak komunikace mezi periferiemi na hardware úrovni. Jedno z povinných rozhraní periférie je AXI, která slouží k její konfiguraci (např. pro UART nastavení délky slova, počet stop bitů atd.), kontrole (např.



Obrázek 5.11: Ukázka rozhraní emulované periférie a typických bloků, z kterých se skládá.

pošli data, proveď reset) a přenosu dat. Druhé povinné rozhraní jsou piny. Periférie však nemá k dispozici jen vstup nebo výstup. Tento signál se skládá ze 3 dalších signálů. Jeden slouží jako vstupní signál (ten čte hodnotu vždy), druhý je pak výstupní signál a třetí signál ovlivňuje, jestli je stav pinu vstup nebo výstup. Je důležité, aby periférie mohly směr signálu ovlivnit, protože například rozhraní I^2C mění směr vodiče SDA podle toho, jestli dané zařízení zprávu přijímá, nebo odesílá. Ostatní signály jsou volitelné. Přerušení je vhodné pro periférie emulující rozhraní, můžeme tak procesor upozornit například na to, že nám došla data nebo že přišla nová zpráva. Signály TX a RX nejsou v tomto návrhu využity a existují pouze pro budoucí kompatibilitu. Jejich zamýšlené využití je pro periférie umožňující posílání velkých množství dat při velké kadenci (signál TX) nebo přesně opačně pro příjem (signál RX). Jedná se tedy o signály typu AXI-Stream, které by byly následně připojené k periférii DMA. Tento způsob je potřebný například pro zařízení typu displej, kde chceme přenést velké množství pixelů do framebuferu v paměti RAM. Obsah periférie nebo rozložení registrů periférie není nijak omezeno a je ponecháno na uživateli.

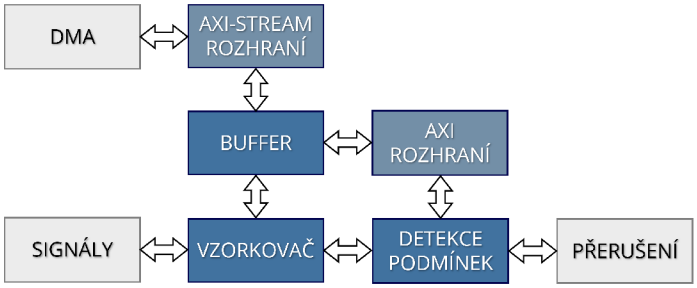
Vytvoření nové funkcionality v FPGA je časově náročné a vyžaduje znalosti, kterými vývojář vestavěných systémů nemusí vždy disponovat. Navržené rozhraní toto bere v potaz a snaží se vyžadovat co nejméně signálů a využívat standardizované rozhraní. Lze tak snadno najít hotová řešení, která buď využívají rozhraní AXI, nebo jim rozhraní AXI lze dodělat. Potřebné úsilí pro rozšíření je tak značně redukováno. Při využívání hotových řešení je ale nutné si uvědomit, že jejich návrh je obvykle optimalizován tak, aby zabral co nejméně místa a byl co nejrychlejší. Když tedy najdeme hotové rozhraní pro SPI, pravděpodobně bude podporovat jen některé módy nebo velikosti slova a nebude možné tyto parametry upravit za běhu. To může být pro potřeby této platformy, kde chceme co největší konfigurovatelnost, značně limitující. Pokud se chceme návrhu hardware vyhnout úplně, součástí návrhu je periférie programovatelné I/O (PIO). Ta ve zkratce slouží pro vytvoření rozhraní skoro na hardware úrovni pomocí skriptu, více je rozebrána v podsekci 5.5. Součástí návrhu jsou také periférie pro rozhraní I^2C, SPI a UART. Návrh každé z nich je odlišný a snaží se demonstrovat možné přístupy k rozšiřitelnosti platformy. Periférie I^2C je tak hotové řešení navržené třetí stranou, SPI je interní periférie SoC a UART je základ navržený třetí stranou a upravený pro potřeby platformy. Na periférii UART je tak ukázáno, jak je možné rozšiřující periférii navrhnout. Detailněji je rozeberán také návrh crossbaru a logického analyzátoru.

Logický Analyzátor

Součástí platformy je pro snažší ladění také jednoduchý logický analyzátor. Existují hotová řešení logických analyzátorů, která by se dala použít. Bohužel je ale většina z nich navržena

jako pomocný obvod pro ladění, který neovlivňuje zbytek platformy a jeho výstup vede na volné piny laděného zařízení. Tento návrh není vhodný pro tuto platformu, která chce logický analyzátor integrovat do pracovního procesu s platformou a ne pouze obsáhnout další nástroj. Bylo tak navrženo vlastní řešení na míru platformě.

Hlavní požadavek na logický analyzátor je zaznamenat průběh signálů na generickém konektoru a tento záznam odeslat uživateli do počítače, kde ho může analyzovat. Uživatel má možnost nastavit podmínku, která spustí záznam. Podmínka se vždy váže na stav signálu a je možné detekovat logickou úroveň signálu (0 nebo 1) nebo změnu logické úrovně (klesající hrana nebo rostoucí hrana). Podmínky lze také kombinovat pomocí globálních operací OR (alespoň jedna podmínka platí) nebo AND (všechny podmínky platí). Pokud jsou podmínky splněny, je spuštěn záznam a pomocí přerušení je na tento fakt upozorněn procesor. Dále je možné nastavit vzorkovací frekvenci, velikost záznamu a případně i posunutí záznamu dopředu nebo dozadu oproti bodu, kdy jsou splněny podmínky pro spuštění záznamu.



Obrázek 5.12: Struktura navrhovaného logického analyzátoru.

Jelikož výstup logického analyzátoru dokáže generovat velké množství dat, musí obsahovat buffer pro jejich uchování. Logický analyzátor má také AXI-Stream výstup, který je připojený k periférii DMA, aby bylo možné všechna data rychle přesunout do paměti RAM. Velikost bufferu může být rovna maximální velikosti záznamu. Použité SoC ale nemá příliš velkou paměť uvnitř FPGA a tak okno bude poměrně malé. Toto můžeme vyřešit tím, že buffer využijeme pouze jako vyrovnávací paměť a periodicky budeme přikazovat DMA přesun dat do paměti RAM. Dosáhneme tak mnohem většího záznamu (desítky až stovky MB oproti desítkám kB). Nicméně, je potřeba zajistit, že DMA přenos a jeho zahájení bude rychlejší než generovaná data.

PIO

Periférie programovatelné I/O (PIO) je periférie umožňující uživateli vytvoření vlastního digitálního rozhraní pomocí skriptu. Motivací pro vlastní návrh byla absence hotových řešení a vytvoření prvního funkčního prototypu této periférie. V rámci práce však nebyl prostor pro plnohodnotný návrh a návrh této periférie je tak silně inspirován návrhem pro čip RP2040 [17] a odlišuje se pouze v detailech.

Hlavní rozdíl je ve struktuře periférie. V tomto návrhu je obsažen pouze jeden procesor, paměť je zvětšena na 64 instrukcí a zároveň jsou zvětšeny i FIFO fronty. Architekturální registry odpovídají původnímu návrhu, to znamená, že při tvorbě programu máme k dispozici dva pracovní registry X a Y, počítadlo instrukcí a dva posuvné registry, jeden pro vstup a druhý pro výstup. Jádro systému je jednoduchý procesor s jednoduchou instrukční sadou o 7 16 bitových instrukcích (tabulka 5.2). Ta se od původního návrhu odlišuje pouze tím, že

neobsahuje instrukci IRQ, zůstává ale zpětně kompatibilní. Následkem ani instrukce WAIT nepodporuje čekání na přerušení. Celá instrukční sada je navržena tak, aby byla co nejkompaktnější, byla výpočetně úplná a umožňovala kontrolu stavu pinu v každém hodinovém cyklu.

Tabulka 5.2: Instrukční sada PIO. P značí polaritu, PP jestli se jedná o PUSH nebo PULL instrukci.

Instrukce	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
JMP	0	0	0	Zpoždění / side-set					Podmínka			Adresa				
WAIT	0	0	1	Zpoždění / side-set					P	Zdroj		Index				
IN	0	1	0	Zpoždění / side-set					Zdroj			Počet bitů				
OUT	0	1	1	Zpoždění / side-set					Cíl			Počet bitů				
PUSHPULL	1	0	0	Zpoždění / side-set					PP	Limit	Blok	0	0	0	0	0
MOV	1	0	1	Zpoždění / side-set					Destinace			Op		Index		
SET	1	1	1	Zpoždění / side-set					Destinace			Data				

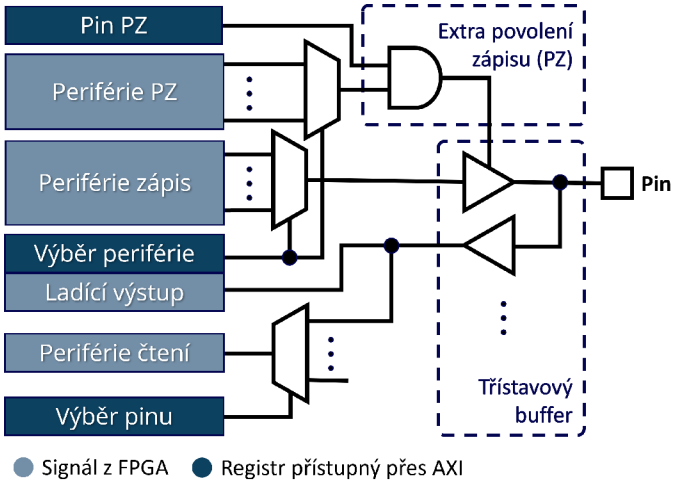
Periférie obsahuje nastavení, která ovlivňují běh programu. Mezi ty hlavní patří auto-push, auto-pull a side-set. Posuvný registr ovlivňujeme instrukcemi IN a OUT, pokud posuvný registr naplníme, začne se přepisovat. Musíme tak hlídat jeho přetečení a periodicky volat instrukci PUSH nebo PULL. Můžeme ale zapnout funkci auto-push/auto-pull, která při překročení hranice automaticky data vymění s FIFO frontou. Ušetříme tak počet potřebných instrukcí. Side-set je pak funkcionalita, která nám umožňuje přidat do každé instrukce změnu stavu pinu. Pokud nastavíme v instrukci JMP side-set na 1 skočí se na adresu a zároveň se nastaví řízený pin na log. 1.

Tyto nastavení lze upravovat přes rozhraní AXI, nelze je ale měnit za běhu. Tento návrh zahrnuje nastavení:

- **zapnutí/vypnutí auto-push a auto-pull**
- **hranice auto-push a auto-pull** — hranice, při které se aktivuje auto-push nebo auto-pull (lze nastavit nezávisle)
- **offset instrukcí** — protože instrukce IN a OUT nemohou adresovat všechny piny, je potřeba relativně posunout nultý pin
- **šířka side-set** — mění bitovou šířku zpoždění vs počet pinů použitelných na side-set
- **offset side-set** — mění pozici, pro kterou side-set bere pin 0

UART

Jako ukázka, jak je možné využít dostupných řešení k rozšíření platformy, je součástí návrhu také periférie UART. Vybere se dostupné jádro této periférie a to bude upraveno tak, aby ho šlo do platformy přidat. Případně bude také upravena implementace, a to takovým způsobem, abychom docílili kýžené konfigurovatelnosti. Pro periférii UART budeme chtít mít možnost nastavit délku zprávy, použitou paritu — lichá, sudá, žádná a počet stop bitů — 1, 1.5, 2 a baudrate.



Obrázek 5.13: Detail funkce crossbaru pro jeden pin.

Digitální Crossbar

Digitální crossbar je navržen tak, aby umožnil připojení N periférií na M pinů s možností měnit toto přiřazení za běhu. Uživatel volí přiřazení pomocí konfiguračních registrů přes rozhraní AXI. Ke konfiguraci má 3 části, ze kterého pinu čte daná periférie, na jaký pin zapisuje která periférie a poté hlavní registr pro povolení zápisu daného pinu. Tento návrh tak umožní číst více periférií z jednoho pinu, zároveň ale umožní zapisovat na jeden pin pouze jedné periferii. Dodatečný registr je v návrhu obsažen z bezpečnostního hlediska. Platforma nastaví výstupy podle uživatelem dodaného nastavení testovaného systému. Zamezí se tak zkratu v případě, pokud by jedna z periférií chtěla chybně zapisovat do vstupního signálu. Z crossbaru vede také ladící výstup, pomocí kterého lze číst aktuální stav pinů po směrování. Tento výstup je dále využitý logickým analyzátorem.

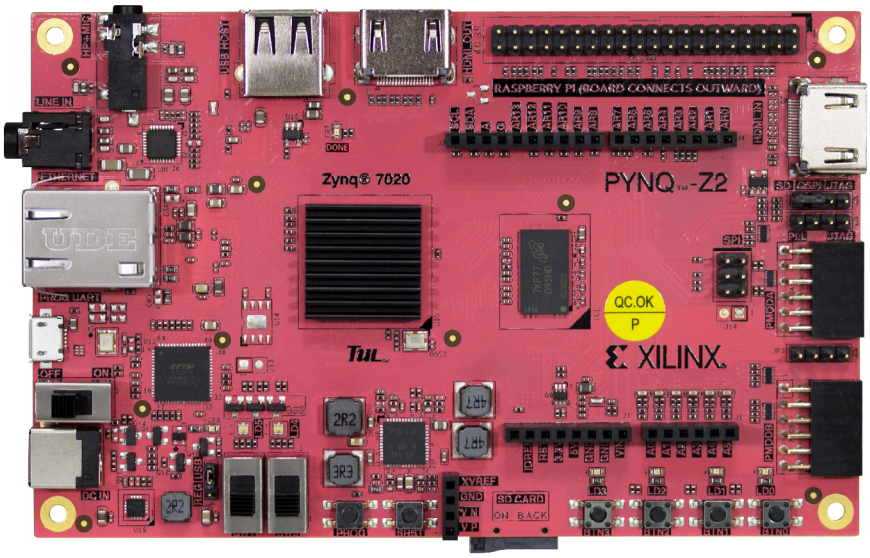
Jelikož architektura FPGA předpokládá přítomnost GPIO periférie ve všech případech, výchozí přiřazení je právě na ní. Zároveň nedává smysl měnit přiřazení GPIO signálů, jsou tak vyřazeny z konfigurační logiky pro snížení potřebného místa.

5.6 Fyzická část platformy

Veškeré požadavky, které pro platformu byly stanoveny, nelze splnit již hotovým řešením. Je tak nutné navrhnout řešení vlastní. Jelikož navrhovat DPS pro tak složité SoC jako je Zynq je komplexní problém, byl vybrán již existující evaluační kit a ten rozšířen o vlastní řešení. Po průzkumu dostupných desek byla zvolena vývojová deska Pynq-Z2. Ta disponuje veškerým potřebným I/O, tedy hlavně gigabitový ethernet, USB-A konektor a dostatečným množstvím přístupných pinů.

Pro tuto vývojovou desku je navržená specializovaná deska, která se nasazuje na vývojovou desku a doplňuje ji o chybějící funkcionality. Desku lze vidět na obrázku 5.15. Funkcionality, které tato deska doplňuje jsou:

- směrování digitálních a anologových signálů včetně mapování funkcionality na výsutpní pin,
- sdružení pinů do jednoho generického konektoru,



Obrázek 5.14: Vývojová deska Pynq-Z2.

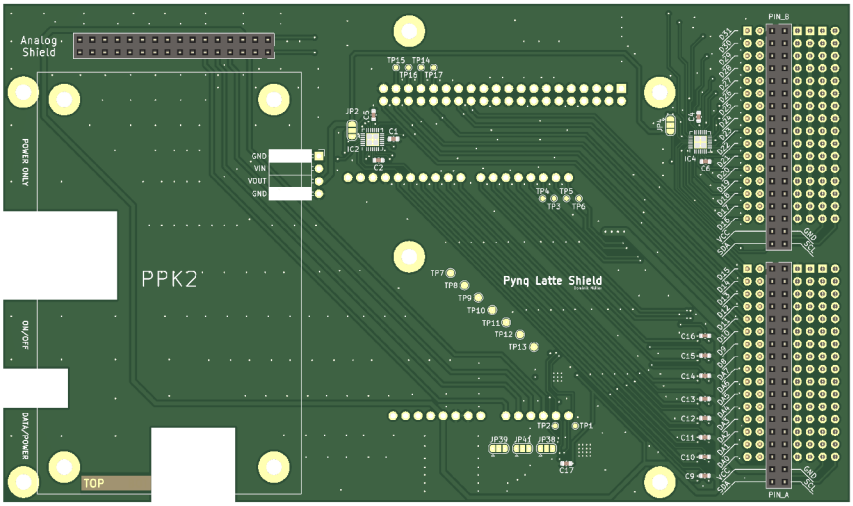
- pull-up a pull-down rezistory,
- doplnění rozhraní pro analogové funkcionality,
- rozšiřující rozhraní pro uživatelské moduly,
- integrace zdroje napájení.

Všechny tyto funkcionality tak integruje do jednoho celku a zároveň ponechává prostor pro další rozšíření. Výsledná schémata navržených desek jsou součástí práce jako přílohy A a B a lze tam nalézt všechny použité součástky. V následujících podkapitolách jsou jednotlivé funkcionality popsány.

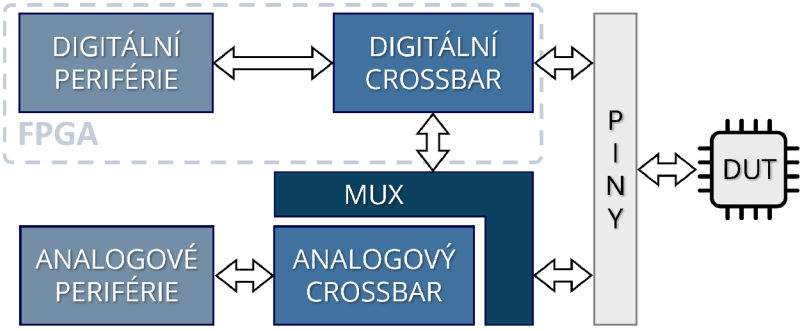
Mapování funkcionalit periférie na libovolný pin

Jelikož plánovaným protikusem platformy je vývojová deska, tedy přímo mikrokontrolér, je vhodné, aby platforma umožňovala přiřadit libovolnou funkcionalitu libovolnému pinu. Stejně tak je tomu také u mikrokontroléru, kde vývojář přiřazuje periférii výstupní pin. Pokud se v průběhu vývoje toto přiřazení změní, platforma musí tuto změnu reflektovat. Platforma miniHIL [4] tento problém řeší pomocí řady pinů a uživatel drátky volí konkrétní zapojení. Požadavek to splní, nicméně značně se tak zvýší šance na špatné zapojení a zátěž na vývojáře. Vývojář musí dávat velký pozor, kam který pin vede a o zapojení více přemýšlet. Zároveň je obtížnější platformu automatizovat, protože změna rozložení pinů vyžaduje fyzický zásah. Je to nevhodné především v počátečních fázích vývoje, kde se rozložení pinů mění velmi často. Tento návrh tak zahrnuje automatické přizpůsobení se definovanému rozložení pinů. Uživatel tak pouze v konfiguračním souboru popíše, jaký pin má kterou funkcionalitu a zbytek zařídí platforma automaticky. Pokud se rozložení změní, uživatel pouze upraví konfigurační soubor.

Podíváme-li se tedy na signály, které musíme přiřazovat, můžeme je rozdělit na analogovou a digitální část. Pro digitální část můžeme problém vyřešit uvnitř FPGA tak, jak



Obrázek 5.15: Rozšiřující deska pro vývojovou desku PYNQ-Z2.



Obrázek 5.16: Diagram připojení signálů z platformy k testovanému systému. Ukazuje způsob jakým je docíleno mapování periférie na jakýkoliv pin.

bylo pospáno v podsekci 5.5. Pro analogovou část musíme vymyslet řešení z dedikovaných komponent. Tento problém lze řešit více způsoby, nicméně ve výsledku vždy implementujeme matici přepínačů. Můžeme tak matici implementovat pomocí diskrétních součástek (tranzistory, relátka) nebo zvolit již hotové řešení. Kvůli nejnižší spotřebě a ceně chceme zvolit již hotové řešení z polovodičové technologie. Hotová řešení pro tento problém existují, pokud však hledáme řešení, která umožní směr signálu v matici v obou směrech, možný výběr se značně zúží. Je tak omezený i možný výběr velikostí matice (typicky násobky 2, tedy 4x8, 8x8 atd.). Maximální dostupná velikost je 16x16. Lze ale skládat matice za sebe a docílit tak větších velikostí. Pokud bychom chtěli docílit 8 analogových funkcionalit na libovolném pinu, můžeme tak zapojit dvě 8x16 matice za sebe. Kromě samotného pole přepínačů obsahují tyto obvody také řídící elektroniku, pomocí které nastavujeme a vyčítáme stav jednotlivých přepínačů. Komunikace s takovou maticí může být realizována pomocí adresy a SR obvodu nebo třeba pomocí I^2C rozhraní. Pro tento návrh potřebujeme docílit možnosti mít na pinu buď digitální signál z FPGA, nebo analogový z externí součástky. Protože digitální signál vedoucí skrz matici nemá žádnou přidanou hodnotu a pouze přináší problémy s přeslechem signálů uvnitř matice (kvůli ostrým hranám digitálního signálu) a zvětšuje požadavky na velikost matice, je směrování digitálního signálu zapojeno dle ob-

rázku 5.16. Do návrhu byla za matici přidána řada diskrétních přepínačů, které vybírají, jestli je na pin připojený analogový nebo digitální signál. Docílíme tak menších přeslechů jednotlivých signálů a zároveň snížíme nároky na velikost matice. Jelikož je úspornější zapojit $N \times N$ matici a N přepínáčů než $N \times 2N$ matici, kde N je počet funkcionalit. Z hlediska funkcionality nás toto omezí pouze tak, že nelze zaráz číst hodnotu pinu digitálně a analogově přímo (nepřímo stále lze). Funkci zápisu to neovlivní vůbec — blokovaná digitální periferie je přesunuta digitálním crossbarem na jiný pin. Kvůli ceně a dostupnosti tohoto druhu matic byla zvolena matice 8x10. Platforma tak umožňuje zapojit až 10 funkcionalit na 8 pinů. Výstup analogového crossbaru je tedy zapojen jen na vybrané vývody a zbytek vývodů nabízí pouze digitální periférie.

Jelikož je analogový crossbar zapojený jako matice přepínačů, umožňuje nám jakékoliv propojení vstupů a výstupů. Z hlediska simulace toho můžeme využít tak, že například výstup jednoho D/A převodníku napojíme na více výstupů. Zároveň je možné zapojit výstup integrovaného D/A převodník na vstup integrovaného A/D převodníku a otestovat tak jejich funkčnost. Je však potřeba speciální validátor, který zajistí, aby uživatelem zvolené zapojení nezpůsobilo poškození platformy.

Generický konektor

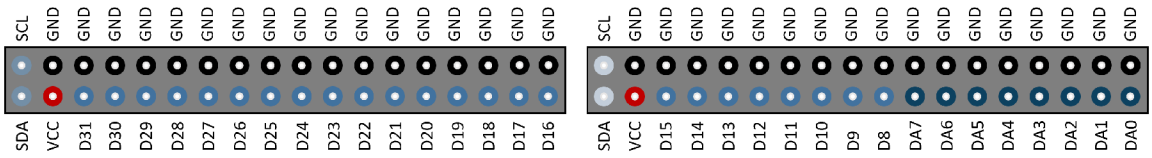
Pro vytvoření konektoru, který bude sloužit pro připojení testovaného zařízení k platformě, se nejdříve musel určit nejvhodnější počet pinů. Byly tak analyzovány často využívané formáty u vývojových kitů. Při analýze byl kladen důraz na počet přístupných pinů a kolik z nich dokáže nabývat analogové funkcionality. Výsledkem je tabulka 5.3. Z té můžeme vyčíst,

Tabulka 5.3: Nejpoužívanější formáty vývojových kitů a jejich vlastnosti.

Formát	**IO**	**ADC**	**DAC**
Arduino Nano	22	8	0
Arduino Uno	14	6	0
Arduino Mega	54	16	0
Nucleo-32	22	8	2
Nucleo-64	86	14	4
Nucleo-144	136	24	2
Raspberry Pi	28	0	0
Raspberry Pico	26	3	0
ESP 32	34	18	2
Nordic	31	7-14	0

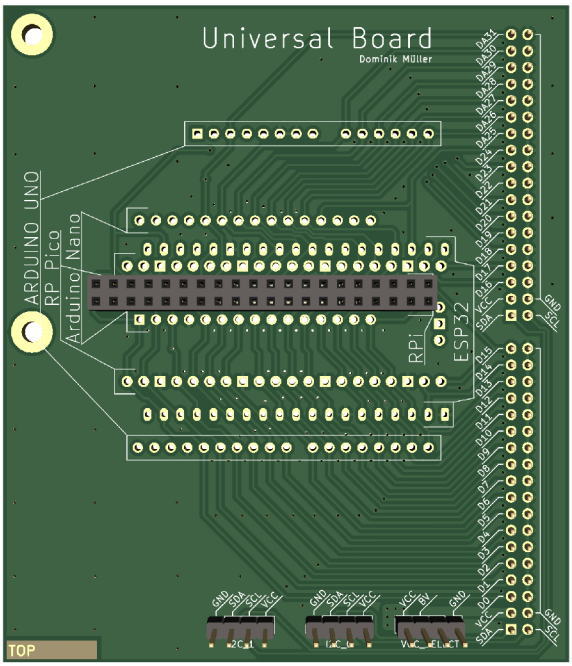
že většina kitů se drží pod 35 pinů a pár z nich má více než jednou tolik pinů. Důležitý je také počet analogových pinů, podle kterých musíme navrhnout analogovou část desky. Pro analogové signály bylo zjištěno, že je zbytečné mít podporu analogové funkcionality na každém pinu a stačí ji mít pouze na části z celkového počtu pinů.

Ačkoliv existují hojně využívané formáty vývojových kitů, nelze zaručit fyzickou kompatibilitu s každým testovaným. Návrh tak zavádí generický konektor jako hlavní rozhraní mezi platformou a testovaným systémem. Je ponecháno na uživateli, aby vytvořil adaptér pro konkrétní testovaný systém. Generický konektor (obrázek 5.17) se skládá z 32 pinů, umožní to plné použití většiny vývojových kitů a u ostatních alespoň menší části z celkového počtu. Zároveň se tento počet zarovná na mocninu dvojky a umožní tak snazší zpracování dat pro logický analyzátor. Počet pinů, které budou mít analogovou funkcionalitu je 8. Na-



Obrázek 5.17: Rozložení pinů generického konektoru, rozteč 2,54 mm.

bídneme tak dostatečný počet analogových pinů pro použití u většiny vestavěných systémů a zároveň udržíme cenu na přijatelné úrovni. Pokud by se ukázal počet analogových pinů jako nedostatečný, lze ho v budoucích návrzích rozšířit přidáním více přepínacích matic a přepínačů. Konektor je koncipován do skupin pinů o 16 signálech, lze tak snadno rozšířit i celkový počet pinů přidáním dalších skupin. Součástí skupiny pinů je také napájení a sběrnice I^{2}C pro případnou konfiguraci.



Obrázek 5.18: Adaptér sdružující nejčastěji používané formáty vývojových kitů.

Jako ukázka využití generického konektoru je součásti návrhu také adaptér, sdružující nejpopulárnější formáty vývojových kitů. Jedná se o formáty: Raspberry Pi, Rapberry Pico, ESP32, Arduino Nano a Arduino Uno.

Pull-up a pull-down rezistory

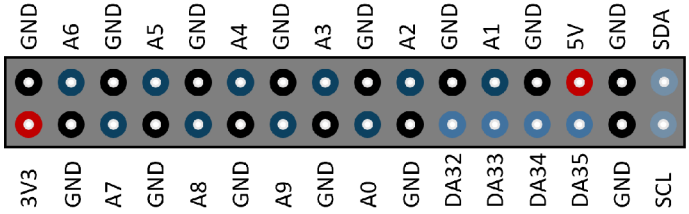
Pro tvorbu rozhraní vestavěných systému potřebujeme mít kontrolu nad logickou úrovní signálu a to včetně pull-up a pull-down rezistorů (viz I^{2}C). Logickou úroveň ovládá logika uvnitř FPGA a vybrané SoC umožňuje specifikovat i připojení pull-up nebo pull-down rezistorů při vytváření binárního souboru. Neumožňuje ale měnit stav těchto rezistorů za běhu FPGA. Pro změnu stavu rezistoru by tak bylo nutné přeprogramovat celé FPGA. To je pro navrhovanou platformu nepřijatelné a je nutné dodat dodatečný obvod. Tento problém

je vyřešen pomocí specializovaných čipů, které slouží pro rozšíření dostupného I/O přes sériové rozhraní. Vzhledem k jejich zamýšlenému použití se tyto čipy chovají podobně, jako periférie GPIO. Umožňují tedy nastavit logickou úroveň nebo připnout pull-up a pull-down rezistor. V tomto návrhu jsou ale využity jen pro dodání pull-up a pull-down funkcionality.

Tím, že použijeme externí čip přes sériové rozhraní, znemožníme stejně rychlé změny stavu rezistorů jako je pro normální log. úroveň. V rámci tohoto návrhu to však není limitující faktor.

Analogová část

Skoro každý moderní mikrokontrolér obsahuje alespoň jednu periférii ADC. Je to způsobeno tím, že vestavěné systémy operují v prostředí, ve kterém snímají spojité veličiny (např. napětí baterie, rychlost motoru atd.). Pokud tak chceme navrhnout platformu pro simulaci okolí vestavěných systémů, nemůžeme opomenout ani analogové signály. V podsekci 5.6 bylo ukázáno, jak se analogové signály dostanou k testovanému zařízení. Nyní bude předvedeno, jak platforma umožňuje takové signály generovat a zpracovávat.



Obrázek 5.19: Rozložení pinů analogového konektoru.

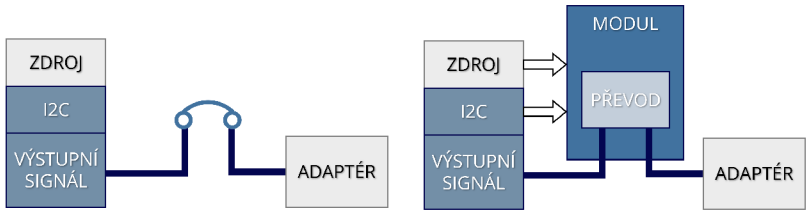
Hlavní spojení s analogovými periférierniemi je další konektor (na obrázku 5.15 v levém horním rohu). Platforma je od analogových periférií oddělená konektorem z toho důvodu, že každý testovaný systém může mít jiné nároky na analogovou část a některé je nemusí využít vůbec. Zároveň se návrh nezabývá návrhem těchto periférií ve velkém detailu, je to tedy také způsob, jak umožnit budoucí rozšíření analogové části. Rozložení jednotlivých pinů můžeme vidět na obrázku 5.19. Konektor obsahuje 10 analogových signálů (A0 – A9), které reprezentují 10 možných analogových funkcionalit a jsou napřímo připojeny k analogové matici. Dále konektor vyhrazuje 4 signály přímo připojené k FPGA (DA32 – DA35). Ty slouží jako pomocné signály pro přenos dat analogovým periférií nebo mohou být použity jako vstup periférie ADC uvnitř SoC. Konektor má také vyhrazenou dedikovanou I^2C sběrnici pro konfiguraci nebo přenos dat. Ostatní piny konektoru slouží pro napájení a lepší integritu signálu.

Nad tímto konektorem tak lze postavit analogové rozšíření, které může přidávat potřebné analogové funkcionality. Hlavní funkcionality uvažované v tomto návrhu jsou ADC jako protikus oproti periférii DAC uvnitř testovaného systému, DAC pro nastavení pomalé analogové hodnoty (např. pro simulaci baterie nebo potenciometru) a generátor analogových funkcí pro simulaci rychlých dějů (např. generace sinusového signálu). Je také možné na konektor přivést signál z úplně jiného zařízení nebo naopak. Můžeme tak připojit například externí osciloskop nebo generátor a platformu využít pouze jako směrovač. Je ale nutné dodržet napětí maximálně 5 V nebo maximální napětí testované platformy.

Pro ověření konektoru je součástí návrhu také jednoduché analogové rozšíření s jedním ADC a DAC. To se skládá ze dvou hotových modulů, které jsou ovládány přes I^2C rozhraní a jsou připojené k analogovému crossbaru.

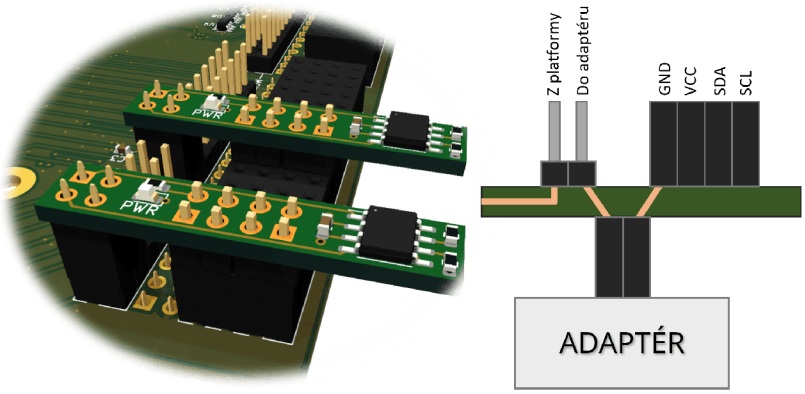
Rozšiřující moduly

Primární rozhraní platformy je vůči vývojovému kitu, motivací platformy je ale také usnadnit vývojáři přechod na specializovaný HIL systém. Aktuálně počítá návrh pouze s běžnými napětími v prostředí číslicových systémů. V pozdějších fází vývoje ale chceme komunikovat s testovaným systémem, který využívá napětí a rozhraní používané mimo DPS testovaného systému. Rozšiřující moduly tak nabízí uživateli možnost, jak se napojit na jednotlivé signály generického konektoru a převést je na signál vhodný pro testovaný systém. Princip modulu je znázorněn na obrázku 5.20.



Obrázek 5.20: Ukázka jak modul zapadá do prostředí platformy. Nalevo platforma bez modulu, napravo po vložení modulu.

Pokud k danému výstupnímu signálu není připojen rozšiřující modul, cesta je propojena a do adaptéru je připojen výstupní signál napřímo. V opačném případě je výstupní signál připojen do rozšiřujícího modulu a až z něho do adaptéru. Do testovaného systému se tak dostane signál modifikovaný rozšiřujícím modulem. Modul může fungovat v obou směrech, tedy jak z adaptéru do platformy, tak i z platformy na adaptér. K převodu může využít buď analogový signál (např. změna napětí sinusového signálu) nebo digitální signál (např. převod UART na RS485). K dispozici má ale i I^2C rozhraní, které může sloužit například pro konfiguraci modulu (např. nastavit míru zesílení sinusového signálu). Jeho konkrétní využití je ale ponecháno na uživateli. Mimo jiné má modul k dispozici samozřejmě i napájení.



Obrázek 5.21: Ukázka fyzického rozložení rozhraní pro moduly.

Konkrétní ukázka rozhraní je na obrázku 5.21. Na ukázce je znázorněno, jak rozhraní vypadá fyzicky a také jak je možné moduly připojit. Je ukázán modul pro 2 signály, vývo-

jář však v tomto aspektu není omezen a může vytvořit modul o 1 – 32 signálech. Modul o jednom signálu už jen s obtížemi umístíme stejně jako na obrázku, je ale stále možné vytvořit modul postavený na výšku. Na pravé straně obrázku je pak znázorněno, jak je rozhraní připojeno ke generickému konektoru a následně do adaptéru. Každá řada obnažených kolíků je připojena k jednomu z pinů generického konektoru. Pomocné signály GND, VCC, SCL a SDA jsou pak sdíleny napříč všemi moduly. Konektor adaptéru využívá, stejně jako generický konektor, rozteč kontaktů 2,54 mm.

Napájecí zdroj

Každý vestavěný systém potřebuje zdroj energie a proto je součástí návrhu této platformy také nastavitelný zdroj napětí. Jelikož je platforma určena hlavně pro připojení vývojových kitů, tak postačí, pokud daný zdroj poskytne napětí alespoň o 3.3 V. Vestavěný systém je také velmi často bateriové zařízení, při jeho vývoji nás tak velmi zajímá jeho spotřeba energie. Z toho důvodu je po použitém zdroji vyžadována schopnost měřit odebíraný proud a to včetně spánkových proudů, které se pohybují v rozmezí mikroampér.

Vývoj vlastního zdroje je nad rámec této práce, bylo tak zvoleno dostupné komerční řešení. Nastavené požadavky výborně splňuje produkt Power Profiler Kit II od společnosti Nordic Semiconductors. Jedná se o produkt pro analýzu příkonu vestavěných zařízení. Nabízí variabilní zdroj napětí (0 – 5 V), který dokáže dodat až 1 A. Součástí je také obvod pro měření spotřeby, který má rozlišení 100 nA nebo 1 mA podle velikosti proudu. Je tedy vhodný jak pro měření spotřeby za běhu procesoru, tak i během spánku. Zdroj je ovládán skrze USB rozhraní a jeho API pro kontrolu je veřejně přístupné. Zároveň se jedná o velmi kompaktní produkt, na obrázku 5.15 je umístěn v levé části označený zkratkou PPK2.

Kapitola 6

Realizace

Tato kapitola se zabývá detaily implementace jednotlivých částí platformy a popisem nástrojů, které byly během vývoje platformy použity. Popis začíná implementací části FPGA, která je základem pro všechny další části. Následně je probráno, jak lze sestavit všechny systémy figurující na fyzickém zařízení platformy. Poté jsou popsány implementační detaily uvnitř těchto systémů a také implementační detaily uživatelského prostředí na počítači. V závěru této kapitoly je představena finální podoba fyzického zařízení a návod, jak platformu připravit na spuštění.

Celý proces realizace týkající se sestavení systému pro SoC nebo vytvoření bitstreamu pro FPGA je postavený na nástrojích poskytovaných výrobcem. Výrobce nabízí různé nástroje pro jednotlivé části SoC:

- Vitis — pro vývoj firmwaru,
- Vivado — pro vývoj bitstreamu pro FPGA,
- Petalinux SDK — pro sestavení operačního systému.

Tyto nástroje mají přiřazenou verzi a jiné verze nástrojů mezi sebou nejsou nutně kompatibilní. Z tohoto důvodu jsou všechny nástroje od tohoto výrobce v této realizaci používány ve verzi 2020.2. Kromě nástrojů výrobce jsou používané také veřejně dostupné nástroje, které jsou blíže popsány v konkrétních sekcích.

Pro vybrané nástroje bylo vytvořeno virtualizované prostředí v prostředí Docker. Je tak možné jednoduše replikovat prostředí pro vývoj, sestavením Docker souborů, které jsou přiložené v odevzdaných souborech. Bylo připraveno prostředí pro vývoj periférií, verifikaci periférií, sestavení systému a vytvoření protokolu pro knihovnu flatbuffers. Všechna tato prostředí byla také použita při vývoji pro automatické sestavení a testování ve verzovací platformě Gitlab.

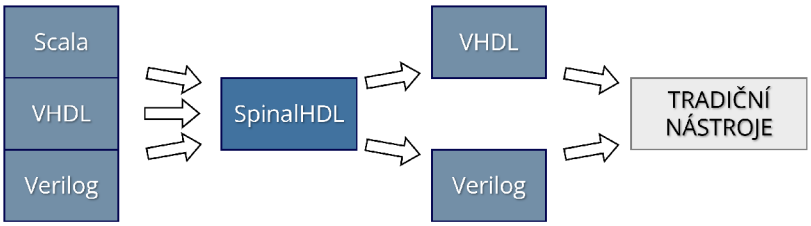
6.1 Vytvoření bitstreamu

V této části je popsáno, jak byly jednotlivé části uvnitř FPGA realizovány a jejich spojení do finálního celku. Výstupem vývoje této části je soubor obsahující konfiguraci FPGA — bitstream, který je použit při programování FPGA. Tento proces má také dva vedlejší výstupy, soubor `.hwh` obsahující popis entit vyskytujících se v návrhu a soubor `.xsa` obsahující popis návrhu včetně nastavení procesoru atd. První ze zmíněných využívá realizovaná platforma pro detekci, jaké periférie jsou uvnitř FPGA a na jakých adresách. Druhý je potřeba pro

vytvoření projektu pro nástroje PetaLinux a Vitis a také pro programování FPGA platformy. Průběh vývoje této části se dá rozdělit na tři fáze. V první fázi se implementovali jednotlivé periférie dle návrhu. V druhé fázi se implementované periférie testovaly, zda se chovají správně a odpovídají návrhu. V poslední fázi pak byly implementované a otestované periférie spojeny do jednoho celku, z kterého se vygeneroval bitstream. V této sekci jsou tyto fáze popsány a na konci jsou také implementační detaily pro vybrané periférie. Realizace pro logický analyzátor a digitální crossbar se neliší od jejich návrhu a jejich implementace probíhala očekávatelně, nejsou tedy zde více popsány.

Implementace periférií

V rámci této práce probíhala implementace jednotlivých částí kompletně mimo prostředí Vivado ve veřejně přístupných nástrojích. Důvodem je, že nástroje výrobce jsou proprietární, špatně přenositelné a vyžadují velké množství úložiště oproti veřejně přístupným nástrojům. Všechny navržené části byly vyvinuty v jazyce SpinalHDL. SpinalHDL je jazyk pro popis hardwaru implementovaný v jazyce Scala. Nejedná se ale o jazyk v tradičním slova smyslu ale spíše o nástavbu nad jazyk Scala. Vývojář má tak přístup k funkcionalitám obou jazyků. Hardware popsaný v jazyce SpinalHDL není syntetizovatelný a je potřeba



Obrázek 6.1: Průběh použití SpinalHDL, jsou sesbírány všechny zdrojové soubory a z nich je vygenerovaná finální entita. Tu pak můžeme použít v klasických RTL nástrojích.

syntetizovatelné soubory vygenerovat. SpinalHDL tak vezme vývojářem vytvořený popis a vygeneruje z něho kód v jazyce VHDL nebo Verilog, který již syntetizovat lze. V této práci se výstup generuje do jazyka VHDL. Nejedná se však o syntézu vysoké úrovně (HLS), popis hardware v jazyce SpinalHDL zůstává na stejné úrovni popisu jako pro VHDL nebo Verilog. Výhoda SpinalHDL je, že do návrhu hardware přináší výhody funkcionálního a objektově orientovaného programování. Lze tak zavést různou míru abstrakce. Například shlukovat jednotlivé signály do skupin (např. rozhraní UART obsahující signály RX a TX) a místo entit vytvářet objekty s atributy a funkcemi, které lze snadno instanciovat a integrovat do vlastního návrhu. Zároveň, tím že máme přístup i k jazyku Scala, můžeme vytvořit generátory, které zredukují množství kódu, který je potřeba napsat. Například v této implementaci je každá periférie obsluhovaná přes rozhraní AXI, které přistupuje ke konfiguračním a kontrolním registrům. Je tak potřeba monotónně popsat adresaci každého registru, přístup ke každému registru atd. Ve SpinalHDL je ale dostupný generátor, kterému specifikujeme jaké registry chceme vytvořit, na jakou adresu je chceme umístit a jaké jsou jednotlivé položky uvnitř těchto registrů. Z tohoto popisu nám pak generátor vygeneruje logiku obsluhující rozhraní s definovanými registry. Ukázka pro jednoduchou násobičku ovládanou přes rozhraní AXI-Lite:

```
case class AxiMul() extends Component {
    // Definice rozhraní entity
```

```
    val io = new Bundle {
        val axi = slave(AxiLite4(AxiLite4Config(
                addressWidth = 3, dataWidth = 32)))
    }

    // Definice AXI registrů
    val busif = BusInterface(io.axi)
    val reg_params = busif.newReg("Parameters")
    val f_a = reg_control.field(
        16 bits, AccessType.WO, 0x00, "Param A")
    val f_b = reg_control.field(
        16 bits, AccessType.WO, 0x00, "Param A")
    val reg_result = busif.newReg("Result")
    val f_o = reg_control.field(
        32 bits, AccessType.RO, 0x00, "Result")
    // Vygenerování AXI rozhraní včetně dokumentace
    busif.accept(HtmlGenerator("axi_pwm.html", "Axi PWM"))

    // Popis chování komponenty
    f_o := f_a * f_b
}
```

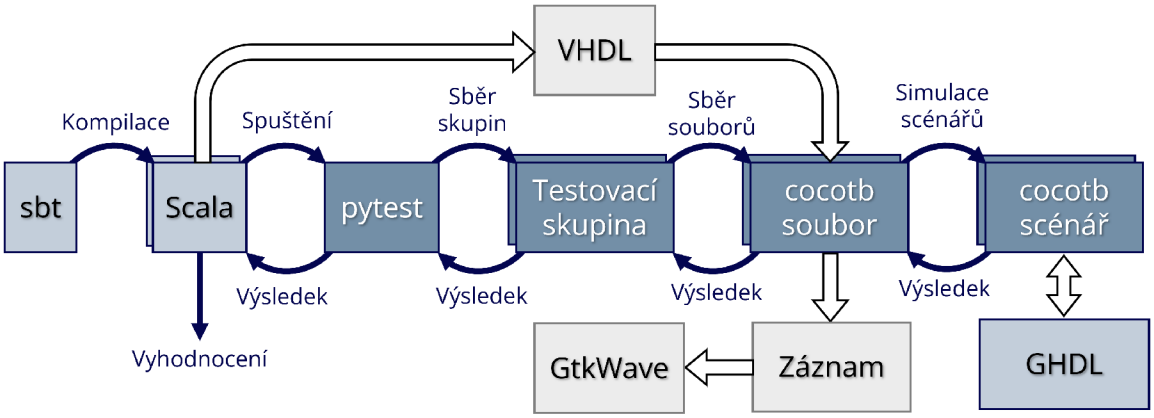
Zároveň lze také přidat další výstup generátoru. Tato práce toho využívá a generuje tak přímo z tohoto popisu podklady pro dokumentaci a jelikož na tyto registry chceme přistupovat z prostředí FreeRTOS, generuje také hlavičkové soubory v jazyce C s definicemi jednotlivých registrů a jejich položek. Všechny tyto přidané funkcionality, ale do návrhu nezavádí žádnou přidanou režii a vše se vyhodnocuje při generaci. SpinalHDL také obsahuje knihovnu základních komponent a rozhraní, které lze při návrhu ihned použít. Můžeme tak například ihned využít frontu FIFO a nemusíme ji ručně implementovat. Z těchto důvodů byl jazyk SpinalHDL upřednostněn oproti tradičnějšímu přístupu. Pro samotnou kompilaci Scala programu a sběr závislostí je použit program sbt.

Funkční verifikace periférií

Po implementaci periférie je vhodné ověřit, že odpovídá stanoveným požadavkům a její chování je validní. Proto využíváme různých verifikačních přístupů a nástrojů, které proces ověření zefektivní. Nástroje výrobce (Vivado) umožňují vytváření testovacích scénářů, ovšem proces jejich vytvoření je zdlouhavý a těžko udržitelný. Byl tak vybrán veřejně dostupný nástroj specializovaný pro spouštění a vyhodnocení testů — cocotb. Nad tímto nástrojem pak bylo vytvořeno testovací prostředí, umožňující snadné a selektivní spouštění testovacích scénářů a přehledné vyhodnocení. V rámci této práce je verifikace periférií pouze základní a ověřuje, že zařízení plní základní funkci. Nevěnuje se detailně všem hraničním scénářům a pro budoucí použití je vhodné provést detailnější verifikaci.

Cocotb je knihovna pro jazyk Python určená pro testování návrhů implementovaných v jazycích VHDL a Verilog. Tato knihovna využívá dostupného API, které simulátory logických obvodů nabízí a vytváří nad nimi abstrakční vrstvu. Tuto vrstvu pak lze uniformně použít pro všechny simulátory z jazyka Python a vytvářet nad ní testovací scénáře. Zároveň knihovna obsahuje pomocné objekty pro práci se signály a podporu pro různé styly verifikace. Velkou výhodou cocotb je také podpora komunity, lze nalézt rozšíření přidáva-

jící podporu pro simulaci rozhraní. Můžeme tak snadno využít tato rozšíření pro testování implementovaných rozhraní (AXI, UART atd.).



Obrázek 6.2: Průběh spuštění a vyhodnocení všech testů.

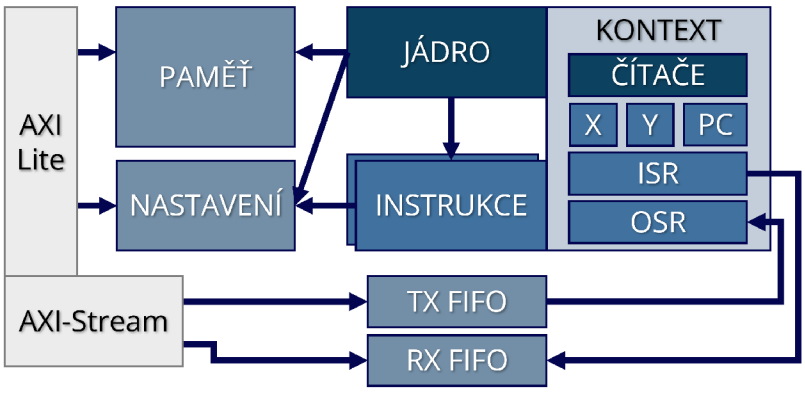
Celý testovací systém je vidět na obrázku 6.2. Tento diagram odpovídá testovacímu prostředí pro všechny implementované periférie. Pro verifikaci periférie je potřeba propojit implementaci v jazyce SpinalHDL s verifikačními nástroji. Jsou tak do testovacího prostředí představeny nástroje umožňující toto propojení. Spuštění testů je zahájeno kompilací Scala souborů a postupným spuštěním všech testovacích objektů. Jednotlivé testovací objekty určují, jaká část návrhu se vygeneruje. Je tak možné testovat část návrhu odděleně. Spuštěný testovací objekt zahájí generaci VHDL souborů a ověří, že je možné takový návrh vygenerovat. V případě úspěchu pokračuje spuštěním programu pytest. Jedná se o program pro sběr, spouštění a vyhodnocení testů napsaných v jazyce Python. Zároveň je to ale i knihovna s nástroji pro podporu testování. Jednotlivé testy, které pytest lokalizuje, jsou brány jako jedna skupina testů. Pod označením skupina je zamýšleno dělení testů podle jejich typu (náhodné, formální atd.), ale není to omezeno. Je tak možné implementovat jednu skupinu jako náhodné testování pomocí jednoho nástroje a druhou skupinu jako formální testování pomocí jiného nástroje. Úkolem skupinového testu je spustit tyto nástroje, sesbírat jejich výsledky, vyhodnotit je a případně také vypsat informaci o chybě. V rámci této práce byly při verifikaci implementovány funkcionální testy s využitím knihovny cocotb. Obrázek 6.2 tak obsahuje průběh konkrétně pro tuto skupinu. Záznam testu generovaný během jednoho souboru obsahuje stav všech signálů v navrhované periférii v průběhu času simulace. Test skupiny lokalizuje všechny soubory obsahující testovací scénáře a soubory jednotlivě spustí. Testy uvnitř skupiny jsou rozděleny do více souborů, protože cocotb spojuje všechny testovací scénáře do jednoho běhu. Pro větší množství testovacích scénářů tak vznikne jeden dlouhý záznam, ve kterém se špatně orientuje. Je to tedy způsob, jak rozdělit testovací scénáře do logických bloků a usnadnit případnou analýzu. Tyto záznamy jsou ve formátu vcd (value change dump) a lze je vizualizovat například pomocí nástroje GtkWave. Kód napsaný v testovacím scénáři pak komunikuje s vybraným simulátorem (v této práci GHDL). Pokud testovací scénář během simulace odhalí chybu, postupně se informace o chybě dopropaguje až do úvodního programu a je prezentována uživateli. V případě, že neměníme popis hardware, lze také generaci VHDL souborů úplně vynechat a spustit pytest napřímo.

Integrace periférií a vytvoření bitstreamu

V okamžiku, kdy jsou všechny periférie implementované a verifikované je nutné je manuálně spojit do jednoho systému. Pro toto spojení se využívá nástroj Vivado. Pomocí něho jsou jednotlivé implementované periférie zabaleny do struktury opakovatelně použitelných bloků — tzv. jednotky duševního vlastnictví (IP Cores). Ty jsou využity při tvorbě blokového schéma celého systému na čipu SoC Zynq. Nástroj Vivado umožňuje vytvoření blokového schéma, kde může uživatel v grafickém rozhraní specifikovat použité entity a vazby mezi nimi. Blokové schéma je automaticky převedeno do jazyka popisující hardware (VHDL nebo Verilog) a následně je provedena syntéza a vytvoření finálního bitstreamu spolu s produkty pro konfiguraci softwarové části.

Obrázek blokového schéma lze nalézt v přílohách práce — příloha C. Z periférií byly do blokového schéma přidány všechny implementované periférie, zároveň byly přidány i periférie I^2C a GPIO implementované výrobcem. Dále byly vyvedeny vývody interní periférie SPI do FPGA. Vývody všech periférií byly připojeny k digitálnímu crossbaru a jejich ovládací rozhraní ke sběrnici AXI. Vzhledem ke struktuře této sběrnice byl použit prvek AXI Interconnect sloužící jako křížový přepínač. Periférie, které obsahují přerušení, byly připojené k řadiči přerušení. Logický analyzátor vyžaduje periférii DMA pro přímý přenos dat do paměti RAM, byla tak připojena na jeho výstup a přes AXI Interconnect na vysokorychlostní port procesoru. Současně byl také připojen signál přerušení do řadiče přerušení. Nakonec byl připojen výstup crossbaru na externí port vedoucí na generický konektor. Součástí blokového schéma jsou také další dvě periférie I^2C pro kontrolu součástek na řidící desce. Takto vytvořené blokové schéma bylo vygenerováno a použito při syntéze. Před syntézou bylo ještě nutné přiřadit externímu portu konkrétní vývody pouzdra FPGA. Platforma využívá k připojení k vývojové desce konektory Raspbery Pi a Arduino Uno, vývody tak byly přiřazeny podle dokumentace od výrobce vývojové desky.

PIO



Obrázek 6.3: Výsledná struktura periférie PIO.

Periférie PIO je implementována podle návrhu a její strukturu lze vidět na obrázku 6.3. Implementace je provedena s předpokladem, že doba pro vykonání instrukce je 1 hodinový takt. Celá periférie je implementována tak, že základem periférie je její jádro, které načítá instrukce z paměti a pokud pro danou instrukci má implementaci, provede ji. Jádro tak zná obecné rozhraní instrukce, ale není přímo navázáno na konkrétní instrukce. Rozhraní pro

instrukci je její operační kód (3 bity), poté logika, která je provedena vždy a logika, která je provedena jen v případě exekuce. Exekuce jádra pak probíhá následovně:

1. načtená instrukce je dekódována,
2. jsou vykonány všechny nepodmíněné logické bloky všech instrukcí,
3. je nastavený side-set, pokud je povolený,
4. pokud instrukce obsahuje zpoždění jádro čeká daný počet taktů,
5. je provedena podmíněná část instrukce.

Všimněme si, že jádro neinkrementovalo programový čítač, to je úkol konkrétní instrukce.

Instrukce jsou v této implementaci brány pouze jako rozšíření jádra periférie. Jednotlivé instrukce jsou tak v implementaci naprosto oddělené od logiky jádra a lze je snadno měnit, implementovat jejich různé verze atd. Každá instrukce musí definovat svůj operační kód a chování. K exekuci má k dispozici celé vybavení jádra (registry, čítače), zde označované jako kontext. Chování je rozděleno na část, která se provádí vždy a část, která se provede pouze tehdy, pokud je instrukce aktivní. Toto rozdělení je potřeba např. pro splnění funkcionality auto-push. Chování funkcionality auto-push odpovídá instrukci PUSH, ale je provedeno automaticky pouze při překročení hranice posuvného registru. V implementaci tak v nepodmíněném bloku instrukce PUSH čeká na to, až bude splněna podmínka. Jinak čeká, dokud není explicitně vyvolána a poté provede podmíněný blok. V implementaci byly implementovány všechny instrukce podle tabulky 5.2. Jejich implementace je přímočará a odpovídá popisu v [17].

Součástí kontextu jsou také posuvné registry — vstupní posuvný registr (ISR) a výstupní posuvný registr (OSR). Ty je možné posunout v jednom hodinovém cyklu o 1 – 32 pozic. Posuvný registr si pamatuje počet posunutí a je saturován na 32 posunutích. Tento počet je pak použit pro funkcionalitu auto-push nebo auto-pull. Oba posuvné registry mají variabilní vstup, respektive výstup. To je zapříčiněno instrukcemi IN a OUT, které mají volitelný zdroj, respektive destinaci. Vzhledem k variabilnímu posunutí s proměnlivými lokacemi je výsledná implementace těchto registrů poměrně komplexní a z celé implementace tak zabírá nejvíce plochy.

Implementace periférie byla ověřena ve verifikačním prostředí. Ověřila se správná funkce všech instrukcí a také schopnost exekuce menšího skriptu:

```
.config:
    autopush 8
    read_pin_offset 1

.program:
    uart_rx:
        wait PINS, False, 0
        set[10] X, 7,
    readloop:
        in PINS, 1
        jmp[6] readloop, X--
        jmp uart_rx
```

Jedná se o jednoduchou implementaci přijímacího kanálu pro UART rozhraní. Na počátku je konfigurace periférie, nastavíme automatické vložení dat do výstupní FIFO fronty po 8 bitech a odsazení prvního čteného pinu. Běh skriptu je pak:

1. počkáme na start bit na pinu 0,
2. počkáme na 10 hod. cyklů a nastavíme registr X na 7,
3. přečteme jeden bit a vložíme ho do ISR,
4. počkáme 6 hodinových cyklů a pokud je $X > 0$ dekrementujeme registr X a skočíme na návěští readloop, v opačném případě pokračujeme další instrukcí,
5. skočíme na začátek programu (přijali jsme jednu zprávu).

Rozhraní se ukázalo jako funkční, v rámci platformy ale nebylo zatím použito. Důvodem je komplexnost začlenění tohoto rozhraní do platformy. Bylo by potřeba implementovat ovladač pro RTOS a především překladač, pro skript zmíněný výše. Jedná se tedy o rozšíření, které může v budoucnu ještě zvýšit modularitu navržené platformy.

UART

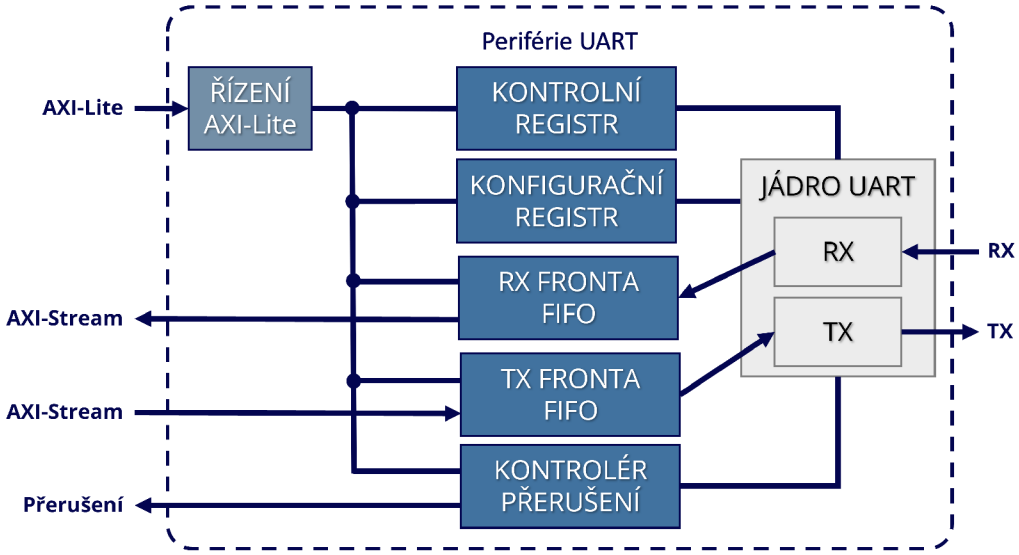
Tato periférie byla v návrhu vybrána jako ukázka, jak lze upravit existující jádro (IP Core) pro potřeby platformy. Popis této periférie tedy bude detailnější. Existující jádro bylo vybráno z knihovny rozhraní jazyka SpinalHDL. Toto jádro se skládá ze dvou nezávislých částí, jedné pro příjem a druhé pro odesílání. Rozhraní pro vstupní i výstupní data je streamového typu (tedy signály TDATA, TVALID, TREADY) se stejným chováním jako AXI-Stream. Obě tyto části sdílejí nastavení jádra, staticky tak nastavíme maximální velikost dat a maximální šířku čítače hodinového signálu. Konkrétní parametry komunikace jako je přenosová rychlost, parita atd. je možné nastavit staticky. Díky přístupu jazyka SpinalHDL lze s tímto nastavením pracovat také jako s registrem a snadno tak vytvořit dynamickou konfiguraci parametrů rozhraní.

Dalším krokem je vytvoření bufferů, které budou zadržovat přijatá data a data k odeslání. Snížíme tak potřebnou režii. Tyto buffery byly implementovány v blokové paměti RAM v FGPA jako fronta FIFO. Nejmenší jednotka alokace takové paměti je 18 KiB pro jeden port a jeden blok o 36 KiB obsahuje dva porty, pro tento návrh jsou potřeba pro každý buffer porty dva. Jeden pro zápis ze strany procesoru a druhý pro čtení ze strany periférie, pro druhý buffer je směr dat naopak. Každý buffer je tak velký 36 KiB pro maximální utilizaci dostupného místa. Maximální velikost dat byla staticky nastavena na 16 bitů, výsledná fronta FIFO tak může pojmout 2304 položek.

V rámci této periférie byly také implementovány porty RX a TX z generického rozhraní periférie. Ty jsou připojeny v případě portu RX k výstupu fronty FIFO a v případě portu TX ke vstupu fronty FIFO. Data z procesoru jsou typicky o šířce 32 bitů, interně v periférii se ale data ukládají jako 16 bitová, byla tak přidána konverze šířky dat.

Do periférie byl přidán také jednoduchý kontrolér přerušení. Ten může vyvolat přerušení v případě, že byla odeslána všechna data, nastala chyba komunikace nebo periférie detekovala přerušení komunikace ze strany přijímače. Všechna tato přerušení lze zapnout, vypnout nebo maskovat, lze také vyčíst jejich stav.

Posledním krokem je přidání AXI-Lite rozhraní, kontrolních a konfigurační registrů. Byl tak zaveden registr pro kontrolu periférie, který umožňuje provést její soft-reset. Dále také konfigurační registr, který umožňuje měnit a vyčítat parametry periférie. Jmenovitě se jedná



Obrázek 6.4: Podoba implementované periférie UART. Modře vyznačené jsou přidané části, šedé je převzaté jádro.

o baudrate, velikost dat v jedné zprávě, použitou paritu a počet stop bitů. Pro přenos dat byl umožněn přístup k frontám FIFO přes rozhraní AXI-Lite. Lze tak vyčíst kolik položek je ve frontě a případně data zapsat nebo vyčíst. Zápis nebo čtení dat automaticky posune frontu. V poslední řadě je pak na rozhraní navázán také kontrolér přerušení, lze tak nastavit jaké přerušení je aktivní a vyčíst jeho stav. Při vyčtení stavu se zároveň automaticky vymaže příznak přerušení.

Takto implementovanou periférii je pak vždy vhodné verifikovat. Dříve a snadněji tak odhalíme možné chyby návrhu, protože jsme schopni během verifikace analyzovat všechny signály uvnitř zařízení. Chyby za běhu periférie na fyzickém zařízení jsou typicky velmi těžké na odhalení a mohou způsobit i zaseknutí celé platformy.

6.2 Sestavení systémů platformy

PetaLinux

V návrhu bylo určeno, že hlavní systém platformy je OS Linux. OS Linux však označuje pouze jádro systému a je potřeba vybrat nebo sestavit vhodnou distribuci, která obsahuje všechny potřebné programy. V rámci této práce byl zvolen nástroj pro sestavení vlastní distribuce doporučený výrobcem — PetaLinux SDK. Jedná se o nástroj udržovaný výrobcem vybraného SoC, který obsahuje nástroje pro sestavení a upravení zavaděče, kernelu systému a také kořenového souborového systému.

Průběh sestavení vlastního systému probíhá tak, že:

1. nastavíme prostředí PetaLinux — nastavení sestavovacího systému, zavaděče, formát výstupního souboru atd.,
2. nastavíme kernel OS Linux — seznam kernelových modulů, parametry systému atd.,
3. upravíme device-tree,

4. nastavíme kořenový souborový systém — seznam programů, uživatele atd.,
5. spustíme sestavení.

Výstupem tohoto procesu je systémový obraz, obsahující všechny potřebné části pro spuštění na cílové platformě. Pomocí PetaLinux SDK byl vytvořen základní projekt, který je již nastavený na výchozí nastavení pro vybrané SoC. Celý projekt je založený na sestavovacím systému Yocto Project. Jednotlivé kroky pro sestavení jsou tak popsány pomocí receptů. V rámci této realizace bylo potřeba nastavit celý proces pro podporu OpenAMP (tento proces je rezebrán v následující podsekci). Dále bylo potřeba přidat podporu USB zařízení pro ovládání napájecího zdroje. To zahrnovalo přidání USB zařízení do device-tree, zapnutí podpory USB zařízení v nastavení kernelu a nastavevní kompilace potřebných kernel modulů pro podporu virtuálního sériového rozhraní, které je použité nad USB pro ovládání napájecího zdroje. Zároveň byly do kořenového souborového systému přidány programy pro podporu vývoje platformy (např. ssh, vim, nettools) a také interpret pro jazyk Python, který je využíván pro implementaci softwarové části. Jelikož softwarová část je závislá na externích modulech, byly zde přidány recepty pro instalaci těchto modulů. Použité nastavení lze detailně prozkoumat v odevzdaných souborech ve složce `os/project-spec/configs/`. Vytvořené moduly pak lze nalézt ve složce `os/project-spec/meta-user/recipes-apps/` a použitý device-tree se nachází v `os/project-spec/meta-user/recipes-bsp/device-tree/files/`.

Tento obraz pak lze vložit na SD kartu a platformu spustit. Při spuštění se nejdříve spustí zavaděč v ROM SoC. Ten podle nastavení pinů lokalizuje zavaděč první fáze, překopíruje ho do paměti RAM a předá mu řízení. Zavaděč první fáze inicializuje a nastaví SoC (případně může také naprogramovat FPGA), lokalizuje zavaděč druhé fáze (U-Boot), ten opět překopíruje do RAM a předá mu řízení. Až tento zavaděč se stará o zavedení hlavního systému OS Linux. Lokalizuje kernel, device-tree a kořenový souborový systém, překopíruje je do paměti RAM a spustí kernel s nastavenými zaváděcími argumenty a adresami, kde se nachází device-tree a kořenový souborový systém. Kernel tyto argumenty zpracuje a spustí uživatelské prostředí. Celý systém běží z paměti RAM, po zavedení tak SD kartu nevyužívá.

Po spuštění uživatelského prostředí se automaticky spustí serverová aplikace a platforma čeká na uživatelský vstup. Při připojení do ethernetového rozhraní systém zkusí, pomocí DHCP klienta, získat IP adresu. Je tak potřeba, aby druhá strana měla dostupný DHCP server. Platformu můžeme připojit buď na lokální síť, kde obdrží IP adresu od routeru, nebo do osobního počítače, kde musí být nainstalovaný a spuštěný DHCP server. Uživatel pak musí zjistit přiřazenou IP adresu svépomocí.

OpenAMP

Tato platforma využívá framework OpenAMP pro nezávislý běh dvou systémů zároveň — v našem případě OS Linux a FreeRTOS. Pro jeho začlenění do platformy je nutné upravit sestavení celého systému. Výrobce tento framework podporuje a jsou dostupné instrukce, jak ho do systému integrovat [5]. Integrace tak postupovala podle těchto instrukcí. Dále jsou popsány pouze provedené úpravy postupu. Pro systém FreeRTOS je vymezená paměť 128 MB, bylo tedy potřeba posunout zaváděcí adresu kernelu na 0x08000000. Takto velká paměť byla zvolena kvůli dostatečnému prostoru pro simulační data. RTOS také potřebuje mít přístup k přerušení, které mohou jednotlivé periférie v FPGA vyvolat. Přiřazení přerušení určitému procesoru řídí globální kontrolér přerušení (GIC). Tento kontrolér je spravován OS Linux. Není tedy vhodné, aby ho druhý systém přenastavoval. Je tak potřeba OS Linux sdělit, že dané přerušení je součástí OpenAMP a patří druhému procesoru.

To se provede zásahem do device-tree, z postupu v [5] se do device-tree přidalo zařízení `remoteproc0`. Tomuto zařízení tak předáme informaci, jaký kontrolér přerušení máme na mysli (`interrupt-parent`) a poté, kterých přerušení se to týká (`interrupts`).

```
remoteproc0: remoteproc@0 {
        compatible = "xlnx,zynq_remoteproc";
        interrupt-parent = <&intc>;
        interrupts = <0 29 4>;
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        memory-region = <&rproc_0_reserved>, <&vdev0buffer>,
                        <&vdev0vring0>, <&vdev0vring1>;
};
```

Jednotlivé položky v atributu `interrupts` jsou:

1. příznak — jestli je přerušení sdílené nebo privátní,
2. index přerušení,
3. typ přerušení — 4 = vyvolej přerušení při log. 1.

Index přerušení zjistíme z technické dokumentace SoC, poté je nutné od indexu z dokumentace odečíst hodnotu 32 a toto číslo dosadit do device-tree. Uvedený index 29 je tedy v dokumentaci označen jako 61.

Když se po této úpravě systém sestaví a spustí, má OS Linux k dispozici 384 MB paměti RAM a obě jádra procesoru, jelikož systém FreeRTOS ještě není spuštěn. K předání jádra procesoru systému FreeRTOS dojde až při jeho spuštění. Pokud FreeRTOS skončí, jádro se opět vrátí pod kontrolu OS Linux. Jelikož OS Linux je v tomto systému master, je možné FreeRTOS ukončit předčasně. To je výhodné, pokud se simulace nechová správně.

FreeRTOS

Implementace systému FreeRTOS proběhla v prostředí Vitis. Pro vytvoření projektu byl potřeba soubor `.xsa`, generovaný při syntéze, ze kterého se získají potřebné soubory pro inicializaci procesoru, ovladače atd. Projekt byl vytvořen z šablony pro prázdnou C++ aplikaci. Dále bylo potřeba do projektu FreeRTOS přidat knihovny libmetal a OpenAMP a upravit nastavení linkovacího skriptu. V tom byly rozšířeny sekce pro haldu a zásobník na hodnotu 0x1000000. Dále je nutné přidat příznak `-DUSE_AMP` do nastavení kompilátoru. Jelikož projekt využívá jazyk C++ je potřeba upravit zdrojové soubory knihoven libmetal a OpenAMP, jinak nelze projekt zkompilovat. Tyto změny jsou prováděny pro verze knihoven metal 2.1 a OpenAMP 1.6. Pro knihovnu metal se jedná o soubor `lib/atomic.h`, zde je potřeba odstranit řádky:

```
20 #elif defined(__cplusplus)
21 # include <atomic>
```

Pro knihvonu OpenAMP se jedná o soubor `lib/include/openamp/virtqueue.h`, zde je potřeba provést tyto změny popsané ve formátu DIFF:

```
@@ -52,11 +52,6 @@
55 -  struct vq_desc_extra {
```

```
56 -      void *cookie;
57 -      uint16_t ndescs;
58 -  };
59 -

@@ -98,7 +93,10 @@
101 -  struct vq_desc_extra vq_descx[0];
96  +  struct vq_desc_extra {
97  +      void *cookie;
98  +      uint16_t ndescs;
99  +  };

@@ -205,7 +203,7 @@
208 -  vqs = (struct virtqueue*)metal_allocate_memory(vq_size);
206 +  vqs = metal_allocate_memory(vq_size);
```

Po těchto úpravách půjde projekt zkompilovat.

6.3 Aplikace uvnitř platformy

Popis této sekce bude jako u návrhu rozdělen do dvou částí a to na popis implementace aplikace na počítači uživatele a aplikace na fyzickém zařízení v prostředí OS Linux. Implementaci aplikace na počítači můžeme dále rozdělit na knihvonu a klientskou aplikaci. Všechny tyto aplikace byly implementovány v jazyku Python a bylo využito značné množství externích knihoven. Důležité knihovny jsou v relevantních sekcích zmíněny a krátce popsány.

Aplikace na počítači

Aplikace na počítači je rozdělena na dvě části — konzolová aplikace a Python modul, který lze importovat a použít pro vlastní účely. Veškerá logika aplikace je implementována v knihovně a konzolová aplikace pouze využívá rozhraní knihovny pro implementaci konkrétních funkcionalit. Pracovní název aplikace je `latte`.

Rozhraní knihovny tvoří jedna třída — `Platform`. Tato třída má na starosti zajištění připojení k fyzickému zařízení a nabízí metody pro naprogramování systémů platformy, vytvoření zařízení a práce s nimi. Pro posílání objemných dat je vyhrazeno speciální API, které umožňuje nahrání těchto souborů do platformy. Vše ostatní je řešeno přes rozhraní GraphQL. Knihovna představuje rozhraní komfortně využitelné z prostředí jazyka Python, toto zpracování pak převádí do jazyka GraphQL a odesíla jej do platformy k vyhodnocení. Pro komunikaci s platformou je na straně klienta použitá knihovna gql, která usnadňuje tvorbu GraphQL dotazů a celkovou komunikaci se serverem GraphQL. Pomocí třídy `Platform`, lze vytvořit zařízení a periférie. Jedná se o speciální třídy, které pomocí API zjistí, jaké položky dané zařízení nabízí a umožňuje k nim snazší přístup přes prostředky jazyka Python.

Samotná konzolová aplikace pak využívá funkcí knihovny a pomocí nich implementuje tyto nástoje:

- aktualizace systému,
- příprava platformy a spuštění interaktivního módu.

Aktualizace systému probíhá pomocí příkazu:

```
$ latte update --target=<ip-adresa> --image=<cesta-k-obrazu>
```

Tento příkaz vyžaduje sestavený obraz systému, který je výstupem nástroje PetaLinux. Nástroj pak nahraje obraz systému do fyzického zařízení a spustí aktualizaci platformy.

Přípravu platformy a následné spuštění interaktivního režimu, lez provést příkazem:

```
$ latte run --target=<ip-adresa> --fw=<simulace> --fpga=<bitstream>
```

Tento příkaz ke své funkci potřebuje informaci jaký kód simulace spustit, potřebuje tedy binární soubor po kompilaci projektu z nástroje Vitis. Dále také potřebuje soubor pro naprogramování FPGA, ten získáme z nástroje Vivado. Tyto soubory nahraje do fyzického zařízení, naprogramuje FPGA a až poté spustí systém RTOS s daným kódem. Pokud vše proběhne úspěšně, je spuštěn interaktivní mód knihovny IPython. V tomto módu má uživatel přístup k interpretu jazyka a funkcím knihovny platformy.

Serverová aplikace

Serverová aplikace na fyzickém zařízení řídí veškerou komunikaci s počítačem. Jedná se o HTTP server běžící na portu 80, nad kterým je ještě použitý framework GraphQL. Implementace HTTP serveru je zajištena pomocí knihovny flask a framework GraphQL pomocí knihovny graphene. Přes knihovnu flask jsou definované podporované cesty HTTP API. Konkrétně se jedná o cesty:

- `/graphql` pro příkazy pro framework GraphQL,
- `/firmware` pro nahrání kódu simulace,
- `/bitstream` pro nahrání bitstreamu pro FPGA,
- `/update` pro nahrání systémového obrazu.

Všechny příkazy měnící nebo dotazující stav platformy jsou prováděny skrz GraphQL a ostatní cesty slouží jen pro nahrání souborů.

Současně se spuštěním serveru je spuštěné také druhé vlákno, které slouží pro detekci USB zařízení. V implementaci je ale využito jen pro detekci napájecího zdroje. Pro ovládání zdroje napájení jeho výrobce poskytuje grafickou aplikaci. Nicméně pro ovládání přímo z programu je potřeba napsat svoji vlastní implementaci. Byl tak vytvořen vlastní ovladač, který umožňuje přepnout mód zdroje a nastavit jeho výstupní napětí. Zdrojové kódy k aplikaci výrobce jsou veřejně dostupné [21], implementační detaily ovladače jsou inspirovány z původní implementace. Funkcionalita pro měření proudu nebyla implementována, lze ji ale snadno přidat stejným způsobem.

Veškerá kontrola platformy je zajištěna pomocí frameworku GraphQL přes dotazy a mutace. Server obsahuje statické objekty, které jsou dostupné vždy (např. `bitstream`, nebo `firmware`). Ty jsou do struktury GraphQL přidané napevno a zároveň mají své konkrétní mutace. Pro ostatní zařízení, které se v platformě vyskytují dynamicky — zařízení a periférie, je struktura GraphQL vytvářena dynamicky podle popisu uvnitř nahraného firmware. Mutace pro tyto zařízení jsou generické a to:

- `setAttribute(name, attribute, value)` — obecné nastavení atributu
- `doAction(name, action)` — vyvolání akce zařízení

Server pak tyto mutace přiřadí konkrétnímu zařízení a vykomunikuje potřebné informace se simulací.

Jeden ze statických objektů je `bitstream` ten udržuje informaci o stavu FPGA a má konkrétní mutaci `DownloadBitstream`. Ta vezme nahraný soubor `.xsa` a vyextrahuje z něj hardwarovou specifikaci a bitstream. Bitstream je přeložen pomocí nástroje výrobce bootgen do binární podoby a přes ovladač fpga_manager je nahrán do FPGA. Hardwarová specifikace je zpracována převzatým skriptem z projektu Pynq a jsou z něho vyextrahovány informace o použitých periferiích. Druhý takový objekt je `firmware`, který udržuje informaci o stavu simulace. Ten má konkrétní mutaci `DownloadFirmware`, která spustí simulaci. Server obsahuje také mutaci `UpdateSystem` pro aktualizaci celého systému. Tento proces je velmi jednoduchý a jeho princip spočívá v tom, že původnímu obraz systému na SD kartě se přidá dovětek `.old`, nahraje se nový obraz a celý systém se restartuje. Všechny ostatní objekty jsou vytvářeny dynamicky.

Pro komunikaci se simulací byl vygenerován serializační a deserializační kód pro jazyk Python. Data serializivána tímto kódem pak jsou přenesena do simulace přes systémové rozhraní ovladače rpmsg. Stejným způsob jsou vyčteny také zprávy ze simulace.

6.4 Implementace simulačního prostředí

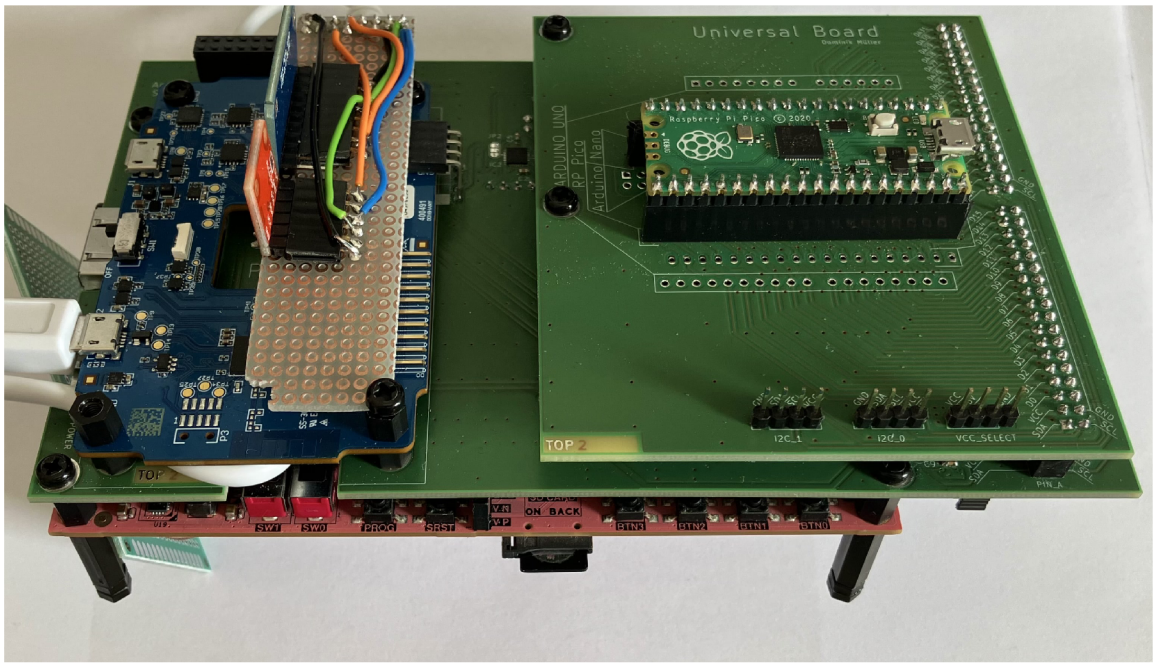
Implementace simulačního prostředí vychází z ukázkového příkladu pro OpenAMP komunikaci. Z příkladu je převzata propagace zpráv z a do OS Linux. V projektu se také používá externí knihovna expected, která usnadňuje řešení chyb v kódu.

Když je FreeRTOS systém spuštěn dojde k inicializaci všech statických zařízení a následně se čeká až OS Linux pošle seznam dostupných periférií uvnitř FPGA. Z tohoto seznamu se vyberou pouze ty periférie pro, které má simulace driver. Jejich seznam společně s dostupnými atributy je odeslán zpět OS Linux. Dostupnost driverů se určuje podle jména IP Core. OS Linux se dále dotáže, jaká zařízení simulace obsahuje a jaké jsou jejich atributy. Další běh simulace se pak odvíjí od implementace uživatele a od podmětů z API. Celkově je řešena implementace simulačního prostředí jako nastínění možné cesty, jakou se vývojář může vydat a snaží se ho tak co nejméně omezovat.

6.5 Fyzická podoba platformy

Součástí návrhu byly tři desky plošných spojů. Hlavní deska a univerzální adaptér byly navrhnuty jako 4 vrstvé DPS, jejich výroba tak byla ponechána externí firmě. Výrobní soubory odeslané externí firmě jsou součástí odevzdaných souborů. Rozšiřující deska pro analogovou část je velmi jednoduchá a byla vyrobena manuálně. Vyrobené desky byly osazeny součástkami a celá platforma byla sestavena a úspěšně oživena. Před sestavením desky je potřeba zkontrolovat, že propojka pro výběr média s obrazem systému je na vývojové desce Pynq Z2 na pozici SD. Sestavení desky pak spočívá pouze v přišroubování distančních šroubků a připojení desek do připravených konektorů. Pro variabilní zdroj napětí je pak ještě nutné natáhnout mikro USB kabel, propojující ho s vývojovou deskou Pynq Z2.

Sestavenou platformu lze vidět na obrázku 6.5. Deska uprostřed je hlavní deska napevno připevněná k vývojové desce. Modrá deska nalevo, do které vede napájecí mikro USB kabel, je variabilní zdroj. Nad ním je volitelná analogová deska s modulem ADS1115 sloužícím jako ADC a MCP4725 plnícím funkci DAC. Napravo se nachází univerzální adaptér, kde je pro ukázku připojený vývojový kit Raspberry Pi Pico. Lokace pro připojení adaptéru obsahuje



Obrázek 6.5: Výsledná podoba fyzického zařízení platformy.

také 2 podpůrné sloupky, které slouží pro snížení namáhání konektoru při častém vyndávání a zandávání vývojových desek. V pravém dolním rohu je také částečně vidět rozhraní pro rozšiřující moduly. Žádné rozšiřující moduly ale neobsahuje a tak jsou kontakty propojeny spojkou.

Pro použití platformy je nutné mít připojený zdroj energie a kabel do ethernetového portu. Uživatel si pro zdroj napájení může zvolit buď mikro USB konektor nebo válcový konektor, do kterého přivede napětí od 7 do 15 V. Při přivedení napětí se rozsvítí červené diody na vývojové desce Pynq Z2, postupem času se zeleně rozsvítí variabilní zdroj. To značí, že byl inicializován USB kontrolér. Po chvíli variabilní zdroj zčervená a platforma je připravena k použití.

Kapitola 7

Vyhodnocení

V této kapitole se nachází konkrétní příklady, jak lze realizované zařízení použít. Zabývá se také vyhodnocením hlavních cílů práce stanovených v kapitole 4 a následně jsou popsány možné způsoby jak platformu dále rozšířit nebo vylepšit.

7.1 Ukázky

Jak přidat periférii?

Pro přidání vlastní periférie je nutné mít implementaci dané periférie a mít ji zabalenou jako IP core pro prostředí Vivado. Poté tuto periférii přidáme do blokového schéma a zapojíme. Je nutné připojit rozhraní AXI k AXI Interconnect a výstup periférie k digitálnímu crossbaru. Pokud periférie obsahuje přerušení, zapojíme ji také ke kontroléru přerušení. Poté blokové schéma syntetizujeme a necháme vygenerovat bitstream.

Dále je nutné implementovat driver pro systém RTOS, který danou periférii dokáže ovládat. V projektu v nástroji Vitis vytvoříme nový soubor, pro danou periférii a v něm vytvoříme novou třídu. Tento driver musí dědit z bázové třídy `Peripheral` a musí implementovat všechny virtuální metody. Možná implementace je ukázaná na tomto jednoduchém příkladu pro periférii UART: Zároveň je potřeba přidanou periférie zaregistrovat do sdružující funkce.

```
#include "driver.hpp"

class Uart: public Peripheral {
public:
    Uart(): Peripheral() { }
    virtual void init(string name, int baseAddress) override {
        // Inicializace periférie
    }
    // Odkaz na jméno IP Core
    std::string getType() { return "uart"; }
    virtual const Attributes getAttributes() override {
        return {
            {"baudrate", Attr<u32>(
                AccessType::RW,
                Uart::getBaudrate, Uart::setBaudrate
            )},
```

```
            {"rxFifoOccupancy", Attr<u32>(
                AccessType::RO, Uart::getRxFifoOccupancy
            )},
            {"reset", Attr<void>(nullptr, nullptr, Uart::reset)},
            // a podobně pro všechny ostatní...
        }
    }

    u32 getBaudrate() { return readReg(BAUDRATE); }
    void setBaudrate(u32 baudrate) { return writeReg(BAUDRATE, baudrate); }
    u32 getRxFifoOccupancy() { return readReg(RX_FIFO_OCCUPANCY); }
    void reset() { return writeReg(RESET); }
};
```

Je tedy nutné implementovat metody `getAttributes` a `getActions` obsahující seznam atributů a akcí. Jednotlivé atributy nebo akce pak potřebují definici metody, která popisuje, jak danou operaci provést.

Jak přidat zařízení?

V projektu v nástroji Vitis přidáme nový soubor pro dané zařízení a v něm vytvoříme novou třídu. Tato třída musí dědit z bázové třídy `Device` a musí implementovat všechny virtuální metody. Zařízení dále může specifikovat atributy a akce, které nabízí a periférie, které využívá. Specifikace atributů a akcí je stejná jako u periférie. Pro specifikaci periférie musí vývojář vytvořit ukazatel na danou periférii a poté v metodě `getPeripherals` vrátit hodnoty z volání funkce `getPeripheral`. Inicializační proceduru je možné vytvořit uvnitř metody `init`. Samotné chování zařízení se implementuje v metodě `behavior`. Každé zařízení pak je potřeba zaregistrovat. Ukázka pro zařízení digitální termostat přes I2C:

```
#include "device.hpp"

class Thermostat: public Device {
public:
    Thermostat(): Device() { }
    virtual void init(string name, vector<string> peripheral) override {
        // Inicializace zařízení
    }

    virtual void behavior() override {
        // Chování zařízení
    }
    std::string getType() { return "thermostat"; }
    virtual const Attributes getAttributes() override {
        // Stejné jako pro periférii
    }

    virtual Peripherals getPeripherals() override {
        return {
            getPeripheral<I2c>(&mpI2c);
        };
```

```
    }
protected:
    I2c *mpI2c;
};
```

Jak přidat real-time test?

Přidání testu v prostředí RTOS je velmi podobné jako přidání zařízení. Jen mizí nutnost specifikovat akce a atributy a místo periférie uvádíme jaké zařízení vyžadujeme v metodě `getDevices`. Ukázka pro test se zařízením termostat:

```
#include "test.hpp"

class MyTest: public Test {
public:
    MyTest(): Test() { }
    virtual void init(vector<string> devices) override {
        // Příprava testu
    }

    virtual void behavior() override {
        // Průběh testu
    }

    std::string getType() { return "my_test"; }

    virtual Devices getDevices() override {
        return {
            getDevice<Thermostat>(&mpThermostat);
        };
    }
protected:
    Thermostat *mpThermostat;
};
```

Jak ovládat simulované prostředí skrze API?

Pro použití API se nejdříve musíme připojit k platformě. Je tak nutné znát dopředu IP adresu platformy a mít připravený bitstream a zkompilovaný kód simulace. Poté můžeme využít interaktivního módu aplikace, ve kterém můžeme ihned s prostředím interagovat. Interaktivní prostředí spustíme příkazem

```
$ latte run --target=<ip-adresa> --fw=<simulace> --fpga=<bitsream>
```

Program se poté připojí k platformě, nahraje soubory, naprogramuje FPGA a zapne simulaci. Následně je uživatel přenesen do interaktivního prostředí jazyka Python3. Následující ukázka ukazuje spuštění testu v tomto prostředí:

```
# Vytvoříme periférii
i2c0 = platform.create_peripheral("i2c")
```

```
# Přiřadíme pin jednotlivým kanálům periférie
i2c0["sda"].pin = 0
i2c0["scl"].pin = 1

# Vytvoříme zařízení termostat
thermostat0 = platform.create_device(
    device="thermostat",
    peripherals=[i2c0]
)

# Změníme cílovou teplotu termostatu
thermostat0["targetTemp"] = 28.2

# Spustíme test
result = platform.run_test("my_test")
```

Ukázka využití při vývoji

Celá platforma byla otestována při experimentech proti reálnému mikrokontroléru. Do univerzálního adaptéru se vložila vývojová deska Raspberry Pi Pico. Do simulace byly implementovány tři jednoduché zařízení e-inkový displej, teploměr a dioda LED. Zároveň byly implementovány také testy pro kontrolu komunikace se zařízeními a kontrolu chování systému. Implementace probíhala stejně jako v ukázkách výše. Simulace se poté zkompilovala a byla přes klientskou aplikaci nahrána do platformy společně s bitstreamem. Platforma se nakonfigurovala pro cílové prostředí. Poté se do mikrokontroléru připojeného k počítači nahrál jeho systém. Ten byl vyvinut na počitači a testování v platformě nijak neovlivnilo jeho vývoj. Po spuštění, mikrokontrolér začne interagovat s okolím a je možné okolí pozorovat a měnit.

7.2 Vyhodnocení požadavků na platformu

V kapitole 4 si práce dala za cíl navrhnout platformu pro vývoj a testování vestavěných systémů, která bude rošířitelná, konfigurovatelná, znovupoužitelná a univerzální. V této části vyhodnotíme, zda došlo k naplnění těchto požadavků.

Rozšířitelnost

Platforma je rozšiřitelná na třech úrovních. Na nejvyšší úrovni jsou simulovaná zařízení (např. teploměr, paměť), která vývojář implementuje na úrovni real-time operačního systému. Na druhé úrovni si volí z předpřipravených periférií nebo implementuje vlastní proti generickému rozhraní. V práci je ukázáno, jak postupovat při jejich integraci, pokud implementujeme řešení vlastní nebo využíváme hotové řešení. Součástí realizace je také funkční periférie PIO, která umožňuje tento proces zjednodušit a implementovat periférii pomocí skriptu. Ta sice není integrována do platformy, ale lze její podporu snadno doplnit. Periférie přímo interagují s testovaným mikrokontrolérem. Pokud by vývojář potřeboval periférie mimo digitální sféru, využívá rozšiřitelnosti platformy na nejnižší úrovni a může si vytvořit

fyzický převodník. Převodník je na jedné straně připojený k platformě přes jednu z dostupných digitálních periférií a na druhé pak k testovanému systému. Můžeme tak snadno přidat bezdrátová rozhraní (WiFi, Bluetooth, ...) nebo rozhraní se speciální fyzickou vrstvou (Ethernet, RS485, ...).

Zároveň návrh připravil rozhraní pro možná budoucí rozšíření. Počítá se s podporou externích USB zařízení pro umožnění automatického testování. Dále bylo připravené rozhraní umožňující uživateli vytvořit vlastní analogové rozšíření.

Konfigurovatelnost

Během vývoje i testování je nutné simulované prostředí upravovat tak, abychom dosáhli kýženého testovacího vektoru. Platforma tak představuje systém akcí a atributů, které vývojáři umožňují definovat parametry zařízení a periférií v testovacím prostředí. Tyto atributy a akce jsou pak automaticky zakomponované do veřejného API platformy a uživatel je může využít při testování nebo experimentech skrz knihovnu v jazyce Python3. Platforma tak přes API umožňuje konfigurovat jak simulované zařízení (teplota, čas, ...), tak i použité periférie, pomocí kterých testovaný mikrokontrolér tyto informace získává a také jejich parametry (baudrate, šířka slova, ...). Uživatel tak má plnou kontrolu nad parametry testovacího prostředí.

Znovupoužitelnost

Definováním pinu mikrokontroléru jako rozhraní platformy, zavedeme co možná nejobecnější rozhraní, na kterém stavíme všechny další elementy testovaného systému. Dostaneme se tak rychle k prvnímu funkčnímu konceptu a rychle se adaptujeme na změny, které nás potkají při vývoji. Navíc jsme schopni inkrementálními úpravami pokrýt testování širokého spektra vestavěných systémů.

Piny mikrokontroléru mohou nabývat více funkcionalit (GPIO, SPI, ADC, ...). K pohodlnému vývoji, bez nutnosti fyzicky měnit zapojení při každé změně rozložení pinů, podporuje platforma mapování jakékoliv digitální funkcionality na jakýkoliv pin. Analogové funkcionality lze mapovat na omezený výběr pinů, obdobně jak tomu je u mikrokontrolérů zvykem. Rozložení pinů mikrokontroléru je specifikováno v konfiguračním souboru, platformu tak lze snadno přenést mezi různými vyvíjenými systémy a změna rozložení se provede automaticky.

Univerzaliza

Vývojář vestavěných systémů využívá při vývoji mnoho nástrojů (např. nástroje pro zprovoznění systému, hledání chyb a validaci jeho chování). Platforma se tak snaží identifikovat ty nejčastější a nejužitečnější nástroje, které vývojář využívá. Tyto nástroje pak integruje do jednoho celku tak, aby vývojáři nabídla univerzální platformu pro vývoj. Výsledná realizace přidává do platformy variabilní zdroj napájení pro napájení testovaného systému. Dále také integruje logický analyzátor, pomocí kterého lze analyzovat děje na pinech testovaného systému.

7.3 Budoucí rozšíření

Budoucí rozšíření, které by umožnilo větší propustnost, je podpora streamování dat z periférií. Jako vhodný případ užití lze uvést simulace displeje. Implementované periférie jsou

na to již připraveny. Jelikož platforma Zynq disponuje pouze dvěma porty pro stream dat, bylo by nutné navrhnout jednotku, která by řídila přepínání kanálů DMA mezi periférie mi. Dále pak navrhnout vhodný způsob přenosu těchto dat do klientské aplikace.

Další možné rozšíření je přidání parciální rekonfigurace. Ta by se uplatnila u volby periférií uživatelem, v FPGA by se přeprogramovala pouze část vyhrazená pro danou periférii a nebylo by nutné přeprogramovat celé FPGA. Pro její přidání by bylo potřeba správně oddělit rozhraní periférie od rekonfigurovatelného regionu a vhodně nastavit vývojové prostředí Vivado.

Pro lepší integraci vývojářem často používaných zařízení, by se mohla implementovat podpora programování a ladění procesoru pomocí JTAG nebo SWD. Toto rozšíření by také umožnilo lepší automatické testování, jelikož by naprogramování testovaného systému nevyžadovalo připojení počítači.

Návrh analogového rozšíření v této práci není příliš detailní. Bylo by tak vhodné vytvořit lepší rozšíření, které by dodávalo více analogových funkcionalit. Např. mít více D/A převodníků nebo také přidat generátor funkcí. Pro toto rozšíření by bylo nutné navrhnout nové analogové rozšíření a zapojit ho do připraveného konektoru. Dále také implementovat jejich ovládání v RTOS.

Kapitola 8

Závěr

Cílem této práce bylo navrhnout a realizovat platformu, umožňující simulaci prostředí vestavěných systémů, která je dosažena emulací periférií uvnitř FPGA. Návrh platformy se soustředil na její snadnou rozšířitelnost, rozsáhlou konfigurovatelnost, univerzalitu a znovupoužitelnost.

Výsledná platforma umožňuje vývojáři vybrat nebo vytvořit zařízení, vyskytující se v okolí vyvíjeného systému a detailně definovat jejich parametry a chování. Simulace okolí mikrokontroléru probíhá přímo proti reálnému mikrokontroléru a je rozdělena na hardwarovou úroveň a na úroveň deterministického real-time prostředí. Použitím tohoto rozhraní odpadají limitace softwarových řešení, ale zároveň zůstává rozhraní dostatečně jednoduché pro snadno dosažitelnou znovupoužitelnost. Navrhovaná platforma tak představuje pohled na testování vestavěných systémů odlišný od běžných řešení.

Konečná podoba platformy se skládá z počítačové aplikace a kompaktního fyzického zařízení. Počítačová aplikace slouží jako vzdálené rozhraní fyzického zařízení a veškerá logika je uvnitř zařízení. Fyzické zařízení je postavené na vývojovém kitu Pynq-Z2 s SoC Zynq 7020. Nad tímto kitem byla navrhnuta vlastní deska dodávající potřebné funkcionality. K této desce pak vývojář připojuje testovaný systém. Pro připojení má 32 vývodů, které jsou přímo napojené na FPGA a 8 z nich také nabízí analogové funkcionality (ADC nebo DAC). Navržená deska si zachovává co možná největší rozšířitelnost a je tak možné analogové rozhraní rozšířit nebo vložit rozšiřující moduly mezi FPGA a výstupní konektor. Mimo jiné platforma obsahuje také variabilní zdroj napájení se schopností měřit běžnou i spánkovou spotřebu testovaného systému a logický analyzátor pro analýzu emulovaných signálů. Platforma tak integrací těchto nástrojů dosahuje univerzality.

Celá platforma je řízena ze systému OS Linux, nicméně pro zajištění real-time simulace obsahuje také systém FreeRTOS. Ten běží s OS Linux v módu asymetrického multiprocesingu. Pro vývojáře je v prostředí FreeRTOS připraveno programové zázemí, ve kterém může implementovat simulované zařízení a periférie. Následně jim definuje atributy a akce, které se automaticky vyextrahují do veřejného API. Rozšíření platformy je možné na více úrovních. První v systému FreeRTOS, druhé na úrovni FPGA a třetí na úrovní hardware pomocí modulů nebo analogových rozšíření. Součástí realizace je také periférie PIO, umožňující vytvoření vlastní digitální periférie pomocí skriptu na hardware úrovni.

Stanovené cíle diplomové práce byly splněny a výsledná platforma slouží jako dobrý základ pro možná budoucí rozšíření.

Literatura

[1] *Glasgow Debug Tool* [online]. [cit. 23. ledna 2022]. Dostupné z: https://github.com/GlasgowEmbedded/glasgow.

[2] *GraphQL* [online]. https://graphql.org/. Dostupné z: https://graphql.org/.

[3] *GreatFET One* [online]. [cit. 23. ledna 2022]. Dostupné z: https://greatscottgadgets.com/greatfet/one/.

[4] *Hardware-in-the-loop Testing* [online]. [cit. 23. ledna 2022]. Dostupné z: https://www.hitex.com/tools-components/test-tools/minihil.

[5] *Libmetal and OpenAMP User Guide* [online]. Dostupné z: https://docs.xilinx.com/v/u/2020.2-English/ug1186-zynq-openamp-gsg.

[6] *Zynq-7000 SoC Product Advantages* [online]. Dostupné z: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[7] Alonso, S., Lazaro, J., Jimenez, J., Muguira, L. a Bidarte, U. Evaluating the OpenAMP framework in real-time embedded SoC platforms. In: *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*. 2021, s. 1–6.

[8] Berger, A. S. *Embedded systems design: An Introduction to Processes, Tools, and Techniques*. CMP Books, 2008. ISBN 1-57820-073-3.

[9] Broekman, B. a Notenboom, E. *Testing embedded software*. Addidon-Wesley, 2003. ISBN 0321159861.

[10] Broesch, J. D., Stranneby, D. a Walker, W. *Digital signal processing: instant access*. Newnes/Elsevier, 2009.

[11] Catsoulis, J. *Designing embedded hardware*. O'Reilly, 2003.

[12] Heath, S. *Embedded Systems Design (Second Edition)*. Newnes, 2003. ISBN 978-0750655460.

[13] Knight, J. Safety critical systems: challenges and directions. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, s. 547–550.

[14] Ledin, J. A. Hardware-in-the-loop simulation. *Embedded Systems Programming*. MILLER FREEMAN INC. 1999, roč. 12, s. 42–62.

[15] Limited, A. *AMBA 4 AXI4-Stream Protocol* [online]. [cit. 26. dubna 2022]. Dostupné z: https://developer.arm.com/documentation/ihi0051/a/.

[16] Limited, A. *AMBA AXI and ACE Protocol Specification* [online]. [cit. 26. dubna 2022]. Dostupné z: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification.

[17] Ltd., R. P. T. *RP2040 Datasheet* [online]. [cit. 23. ledna 2022]. Dostupné z: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf.

[18] Luis, G. a M., F. J. *Behavioral modeling for embedded systems and technologies: applications for design and implementation.* IGI Global (701 E. Chocolate Avenue, Hershey, Pennsylvania, 17033, USA), 2010.

[19] Marwedel, P. *Embedded system design: embedded systems foundations of cyber-physical systems, and the Internet of Things.* Springer, 2021. ISBN 978-3-030-60909-2.

[20] Muresan, M. a Pitica, D. Software in the Loop environment reliability for testing embedded code. In: *2012 IEEE 18th International Symposium for Design and Technology in Electronic Packaging (SIITME).* 2012, s. 325–328.

[21] NordicSemiconductor. [online]. Dostupné z: https://github.com/NordicSemiconductor/pc-nrfconnect-ppk.

[22] Reghenzani, F., Massari, G. a Fornaciari, W. The real-time linux kernel: A survey on Preempt_RT. In:. únor 2019, s. 1–36.

[23] Roshandel Tavana, N. a Dinavahi, V. A General Framework for FPGA-Based Real-Time Emulation of Electrical Machines for HIL Applications. *IEEE Transactions on Industrial Electronics.* 2015, roč. 62, č. 4, s. 2041–2053.

[24] Růžička, R. *Vestavné systémy.* Vyskoé učení technické v Brně, Fakulta informačních technologií.

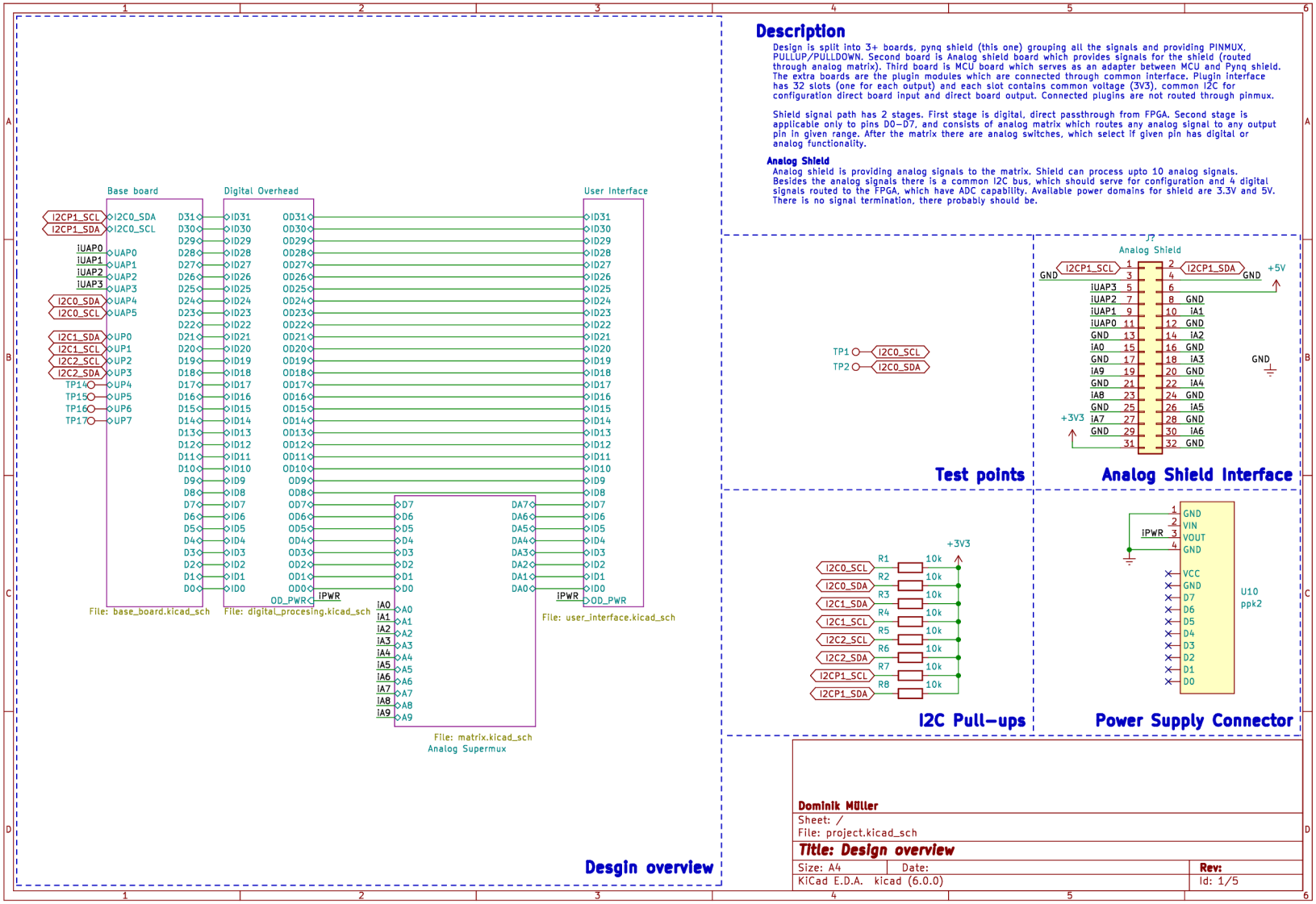
[25] Stankovic, J. A. Real-time and embedded systems. In: *ACM Computing Surveys.* 1996, s. 205–208.

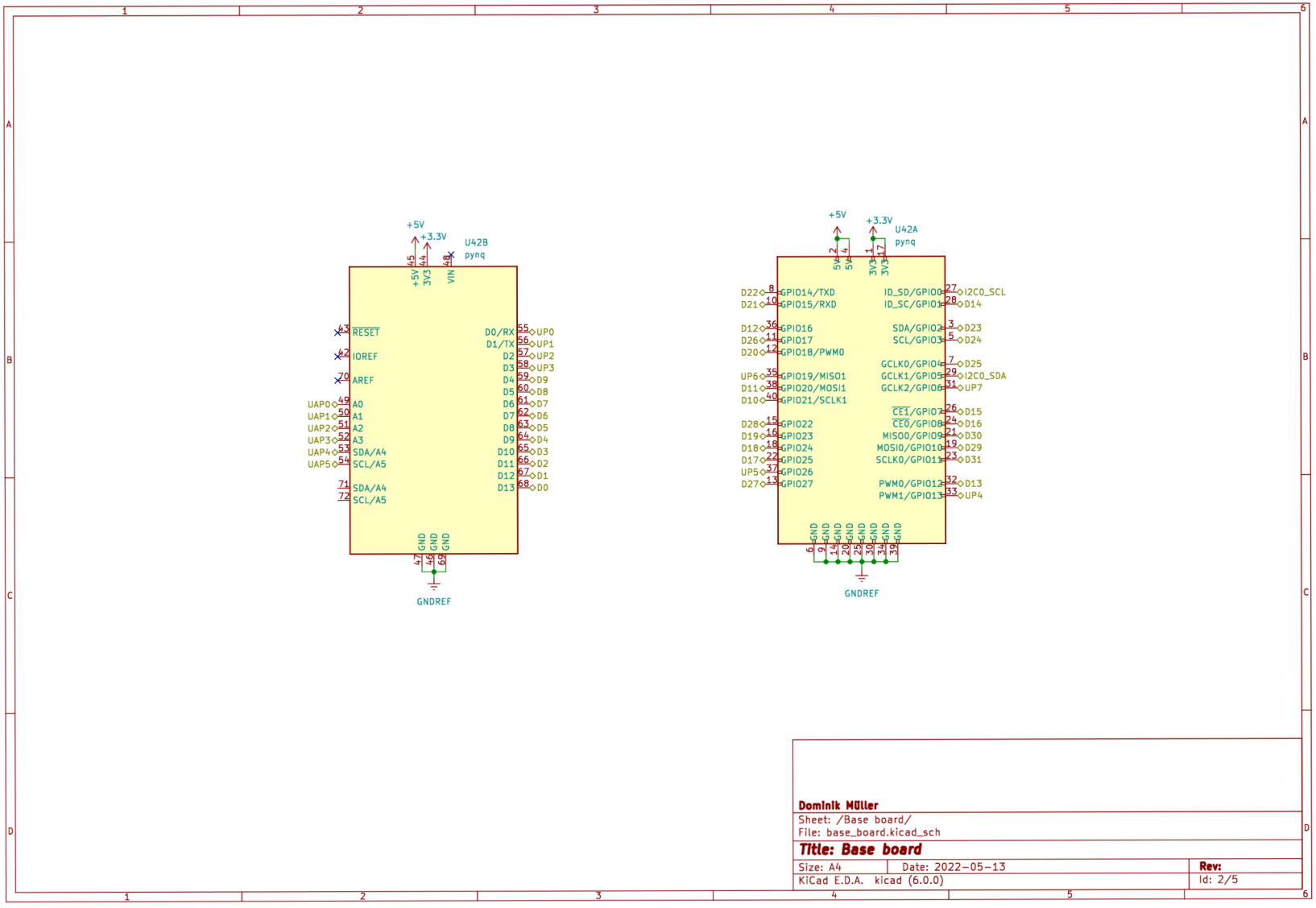
[26] Vardhan, H., Akin, B. a Jin, H. A Low-Cost, High-Fidelity Processor-in-the Loop Platform: For Rapid Prototyping of Power Electronics Circuits and Motor Drives. *IEEE Power Electronics Magazine.* 2016, roč. 3, č. 2, s. 18–28.

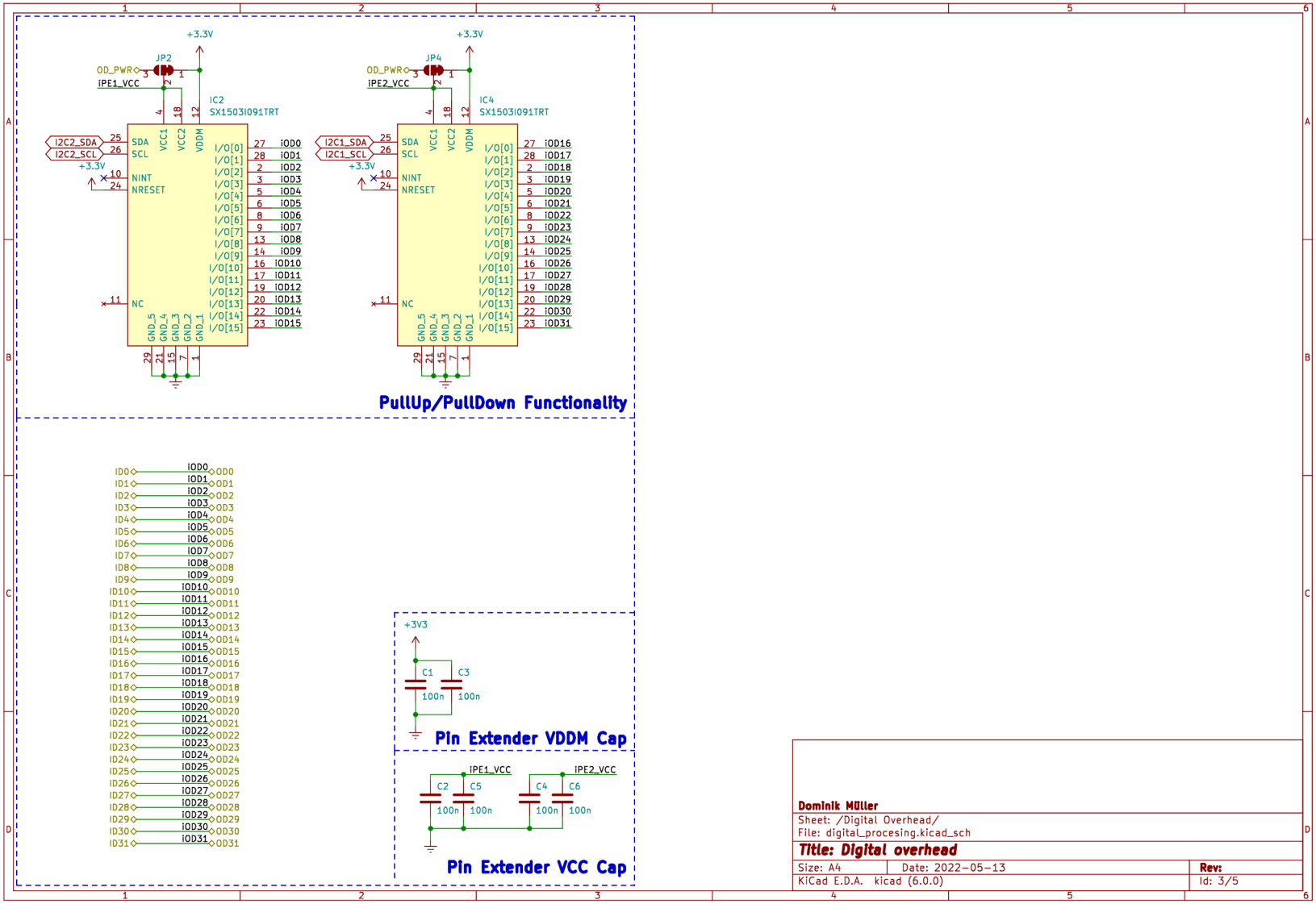
[27] Vardhan, H., Akin, B. a Jin, H. A Low-Cost, High-Fidelity Processor-in-the Loop Platform: For Rapid Prototyping of Power Electronics Circuits and Motor Drives. *IEEE Power Electronics Magazine.* 2016, roč. 3, č. 2, s. 18–28.

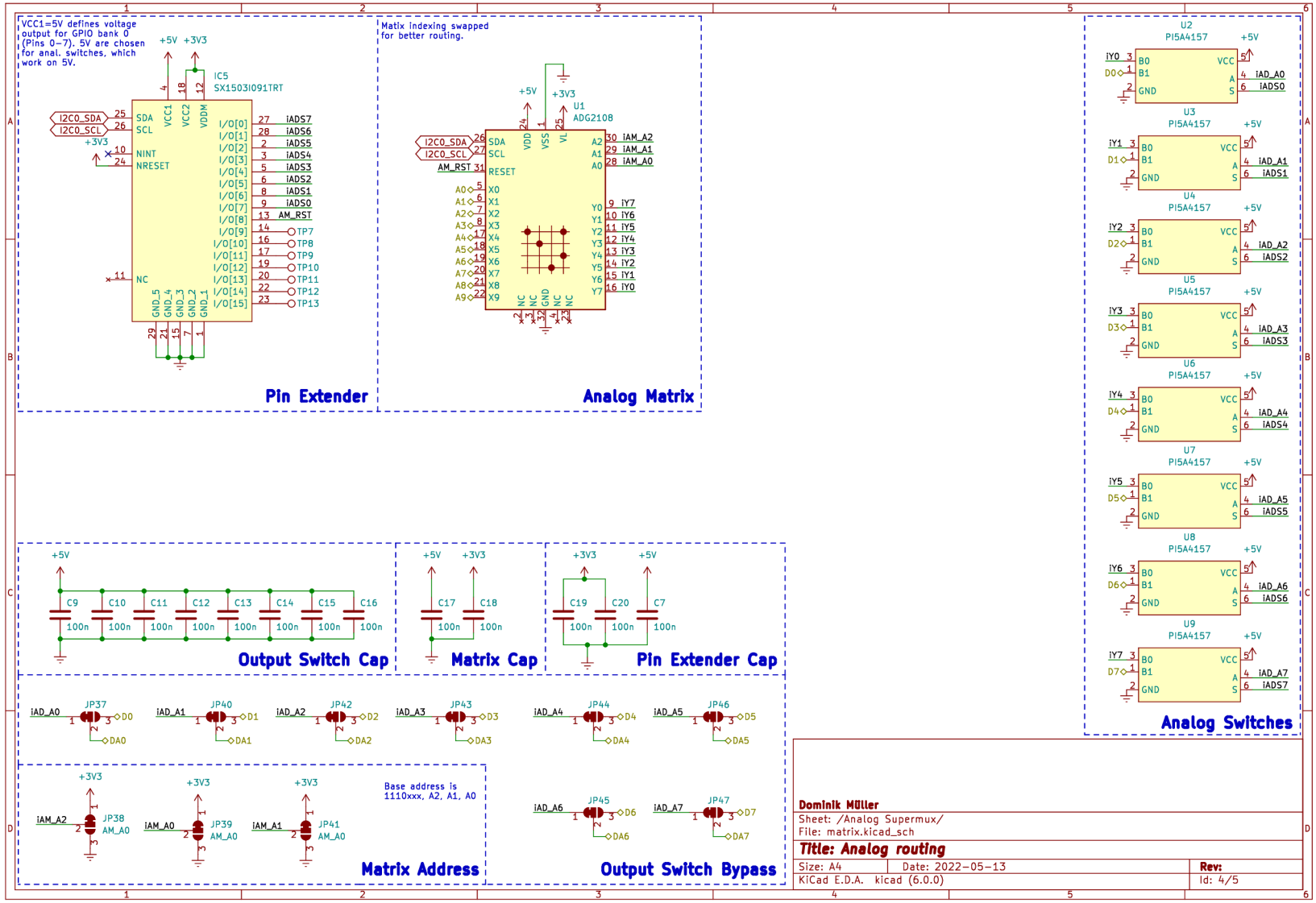
Příloha A

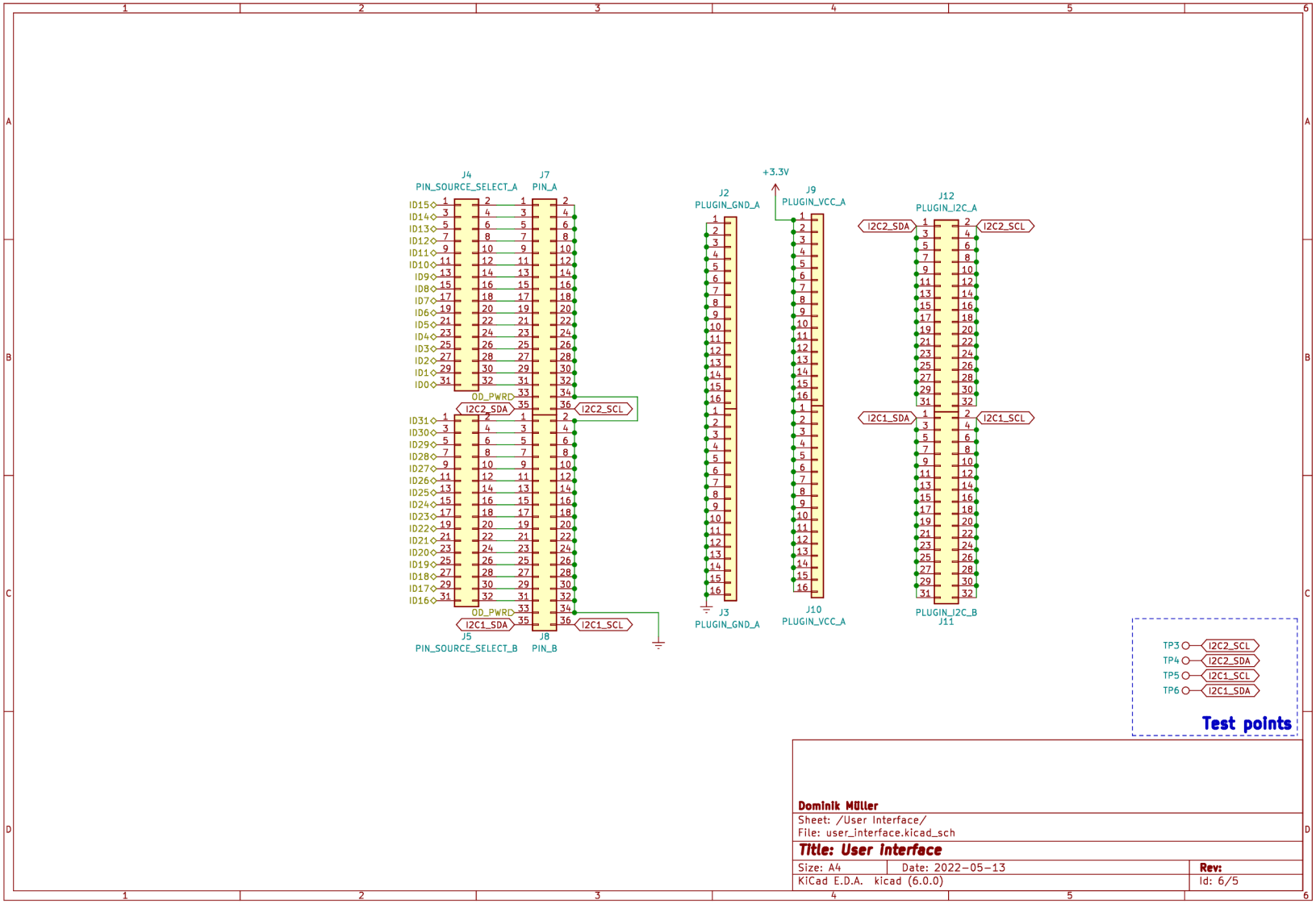
Schéma hlavní desky platformy


Base board
Digital Overhead
User Interface
I2CP1_SCL
I2CP1_SDA
I2C0_SDA
I2C0_SCL
iUAP0
iUAP1
iUAP2
iUAP3
UAP0
UAP1
UAP2
UAP3
UAP4
UAP5
I2C0_SDA
I2C0_SCL
I2C1_SDA
I2C1_SCL
I2C2_SCL
I2C2_SDA
UP0
UP1
UP2
UP3
UP4
UP5
UP6
UP7
TP14
TP15
TP16
TP17
D31
D30
D29
D28
D27
D26
D25
D24
D23
D22
D21
D20
D19
D18
D17
D16
D15
D14
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
ID31
ID30
ID29
ID28
ID27
ID26
ID25
ID24
ID23
ID22
ID21
ID20
ID19
ID18
ID17
ID16
ID15
ID14
ID13
ID12
ID11
ID10
ID9
ID8
ID7
ID6
ID5
ID4
ID3
ID2
ID1
ID0
OD31
OD30
OD29
OD28
OD27
OD26
OD25
OD24
OD23
OD22
OD21
OD20
OD19
OD18
OD17
OD16
OD15
OD14
OD13
OD12
OD11
OD10
OD9
OD8
OD7
OD6
OD5
OD4
OD3
OD2
OD1
OD0
OD_PWR
iPWR
File: base_board.kicad_sch
File: digital_procesing.kicad_sch
D7
D6
D5
D4
D3
D2
D1
D0
iA0
iA1
iA2
iA3
iA4
iA5
iA6
iA7
iA8
iA9
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
DA7
DA6
DA5
DA4
DA3
DA2
DA1
DA0
File: matrix.kicad_sch
Analog Supermux
File: user_interface.kicad_sch
Desgin overview
Description
Design is split into 3+ boards, pynq shield (this one) grouping all the signals and providing PINMUX, PULLUP/PULLDOWN. Second board is Analog shield board which provides signals for the shield (routed through analog matrix). Third board is MCU board which serves as an adapter between MCU and Pynq shield. The extra boards are the plugin modules which are connected through common interface. Plugin interface has 32 slots (one for each output) and each slot contains common voltage (3V3), common I2C for configuration direct board input and direct board output. Connected plugins are not routed through pinmux.
Shield signal path has 2 stages. First stage is digital, direct passthrough from FPGA. Second stage is applicable only to pins D0–D7, and consists of analog matrix which routes any analog signal to any output pin in given range. After the matrix there are analog switches, which select if given pin has digital or analog functionality.
Analog Shield
Analog shield is providing analog signals to the matrix. Shield can process upto 10 analog signals. Besides the analog signals there is a common I2C bus, which should serve for configuration and 4 digital signals routed to the FPGA, which have ADC capability. Available power domains for shield are 3.3V and 5V. There is no signal termination, there probably should be.
TP1
TP2
I2C0_SCL
I2C0_SDA
Test points
J?
Analog Shield
GND
I2CP1_SCL
I2CP1_SDA
+5V
iUAP3
iUAP2
iUAP1
iUAP0
iA1
iA2
iA3
iA4
iA5
iA6
iA7
iA8
iA9
+3V3
Analog Shield Interface
R1
R2
R3
R4
R5
R6
R7
R8
10k
I2C1_SDA
I2C1_SCL
I2C2_SCL
I2C2_SDA
I2C Pull-ups
VIN
VOUT
U10
ppk2
VCC
Power Supply Connector
Dominik Müller
Sheet: /
File: project.kicad_sch
Title: Design overview
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 1/5


+5V
+3.3V
U42B
pynq
+5V
3V3
VIN
RESET
IOREF
AREF
A0
A1
A2
A3
SDA/A4
SCL/A5
SDA/A4
SCL/A5
D0/RX
D1/TX
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
UP0
UP1
UP2
UP3
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
UAP0
UAP1
UAP2
UAP3
UAP4
UAP5
GND
GND
GND
GNDREF
+5V
+3.3V
U42A
pynq
5V
5V
3V3
3V3
GPIO14/TXD
GPIO15/RXD
GPIO16
GPIO17
GPIO18/PWM0
GPIO19/MISO1
GPIO20/MOSI1
GPIO21/SCLK1
GPIO22
GPIO23
GPIO24
GPIO25
GPIO26
GPIO27
ID_SD/GPIO0
ID_SC/GPIO1
SDA/GPIO2
SCL/GPIO3
GCLK0/GPIO4
GCLK1/GPIO5
GCLK2/GPIO6
CE1/GPIO7
CE0/GPIO8
MISO0/GPIO9
MOSI0/GPIO10
SCLK0/GPIO11
PWM0/GPIO12
PWM1/GPIO13
I2C0_SCL
D14
D23
D24
D25
I2C0_SDA
UP7
D15
D16
D30
D29
D31
D13
UP4
D22
D21
D12
D26
D20
UP6
D11
D10
D28
D19
D18
D17
UP5
D27
UP2
UP1
UP0
UP3
GND
GNDREF
Dominik Müller
Sheet: /Base board/
File: base_board.kicad_sch
Title: Base board
Size: A4
Date: 2022-05-13
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 2/5

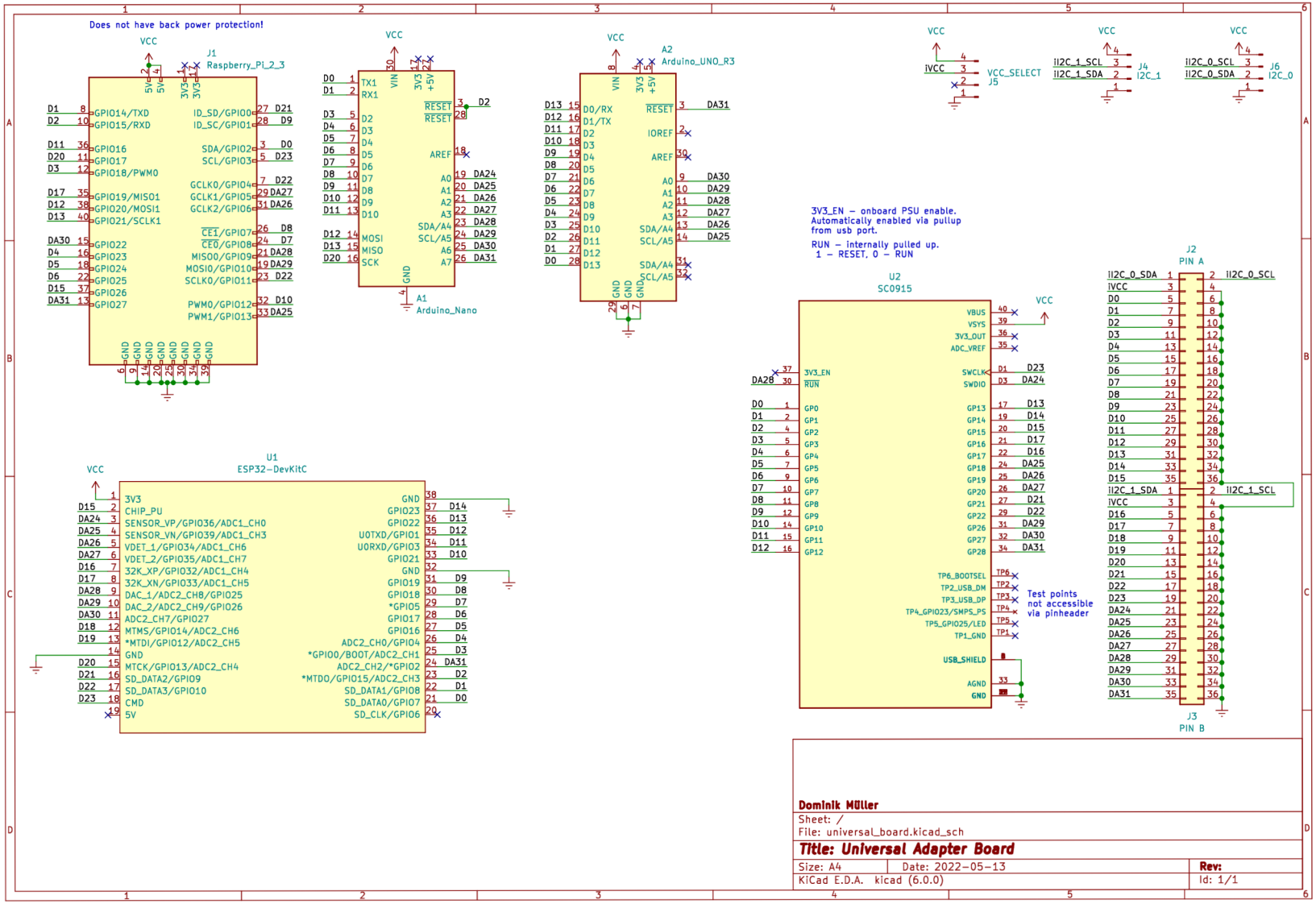

+3.3V
JP2
OD_PWR
IPE1_VCC
IC2
SX1503I091TRT
I2C2_SDA
I2C2_SCL
SDA
SCL
VCC1
VCC2
VDDM
NINT
NRESET
NC
GND_5
GND_4
GND_3
GND_2
GND_1
I/O[0]
I/O[1]
I/O[2]
I/O[3]
I/O[4]
I/O[5]
I/O[6]
I/O[7]
I/O[8]
I/O[9]
I/O[10]
I/O[11]
I/O[12]
I/O[13]
I/O[14]
I/O[15]
iOD0
iOD1
iOD2
iOD3
iOD4
iOD5
iOD6
iOD7
iOD8
iOD9
iOD10
iOD11
iOD12
iOD13
iOD14
iOD15
JP4
IPE2_VCC
IC4
SX1503I091TRT
I2C1_SDA
I2C1_SCL
iOD16
iOD17
iOD18
iOD19
iOD20
iOD21
iOD22
iOD23
iOD24
iOD25
iOD26
iOD27
iOD28
iOD29
iOD30
iOD31
PullUp/PullDown Functionality
ID0
ID1
ID2
ID3
ID4
ID5
ID6
ID7
ID8
ID9
ID10
ID11
ID12
ID13
ID14
ID15
ID16
ID17
ID18
ID19
ID20
ID21
ID22
ID23
ID24
ID25
ID26
ID27
ID28
ID29
ID30
ID31
OD0
OD1
OD2
OD3
OD4
OD5
OD6
OD7
OD8
OD9
OD10
OD11
OD12
OD13
OD14
OD15
OD16
OD17
OD18
OD19
OD20
OD21
OD22
OD23
OD24
OD25
OD26
OD27
OD28
OD29
OD30
OD31
+3V3
C1
C3
100n
Pin Extender VDDM Cap
IPE1_VCC
IPE2_VCC
C2
C5
C4
C6
Pin Extender VCC Cap
Dominik Müller
Sheet: /Digital Overhead/
File: digital_procesing.kicad_sch
Title: Digital overhead
Size: A4
Date: 2022-05-13
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 3/5


VCC1=5V defines voltage output for GPIO bank 0 (Pins 0-7). 5V are chosen for anal. switches, which work on 5V.
IC5
SX1503I091TRT
I2C0_SDA
I2C0_SCL
AM_RST
Pin Extender
Matix indexing swapped for better routing.
U1
ADG2108
Analog Matrix
U2
PI5A4157
Analog Switches
Output Switch Cap
Matrix Cap
Pin Extender Cap
Base address is 1110xxx, A2, A1, A0
Matrix Address
Output Switch Bypass
Dominik Müller
Sheet: /Analog Supermux/
File: matrix.kicad_sch
Title: Analog routing
Size: A4
Date: 2022-05-13
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 4/5


J4
PIN_SOURCE_SELECT_A
J7
PIN_A
ID15
ID14
ID13
ID12
ID11
ID10
ID9
ID8
ID7
ID6
ID5
ID4
ID3
ID2
ID1
ID0
OD_PWRD
I2C2_SDA
I2C2_SCL
ID31
ID30
ID29
ID28
ID27
ID26
ID25
ID24
ID23
ID22
ID21
ID20
ID19
ID18
ID17
ID16
OD_PWRD
I2C1_SDA
I2C1_SCL
J5
PIN_SOURCE_SELECT_B
J8
PIN_B
J2
PLUGIN_GND_A
+3.3V
J9
PLUGIN_VCC_A
J12
PLUGIN_I2C_A
I2C2_SDA
I2C2_SCL
I2C1_SDA
I2C1_SCL
J3
PLUGIN_GND_A
J10
PLUGIN_VCC_A
PLUGIN_I2C_B
J11
TP3
TP4
TP5
TP6
I2C2_SCL
I2C2_SDA
I2C1_SCL
I2C1_SDA
Test points
Dominik Müller
Sheet: /User Interface/
File: user_interface.kicad_sch
Title: User Interface
Size: A4
Date: 2022-05-13
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 6/5

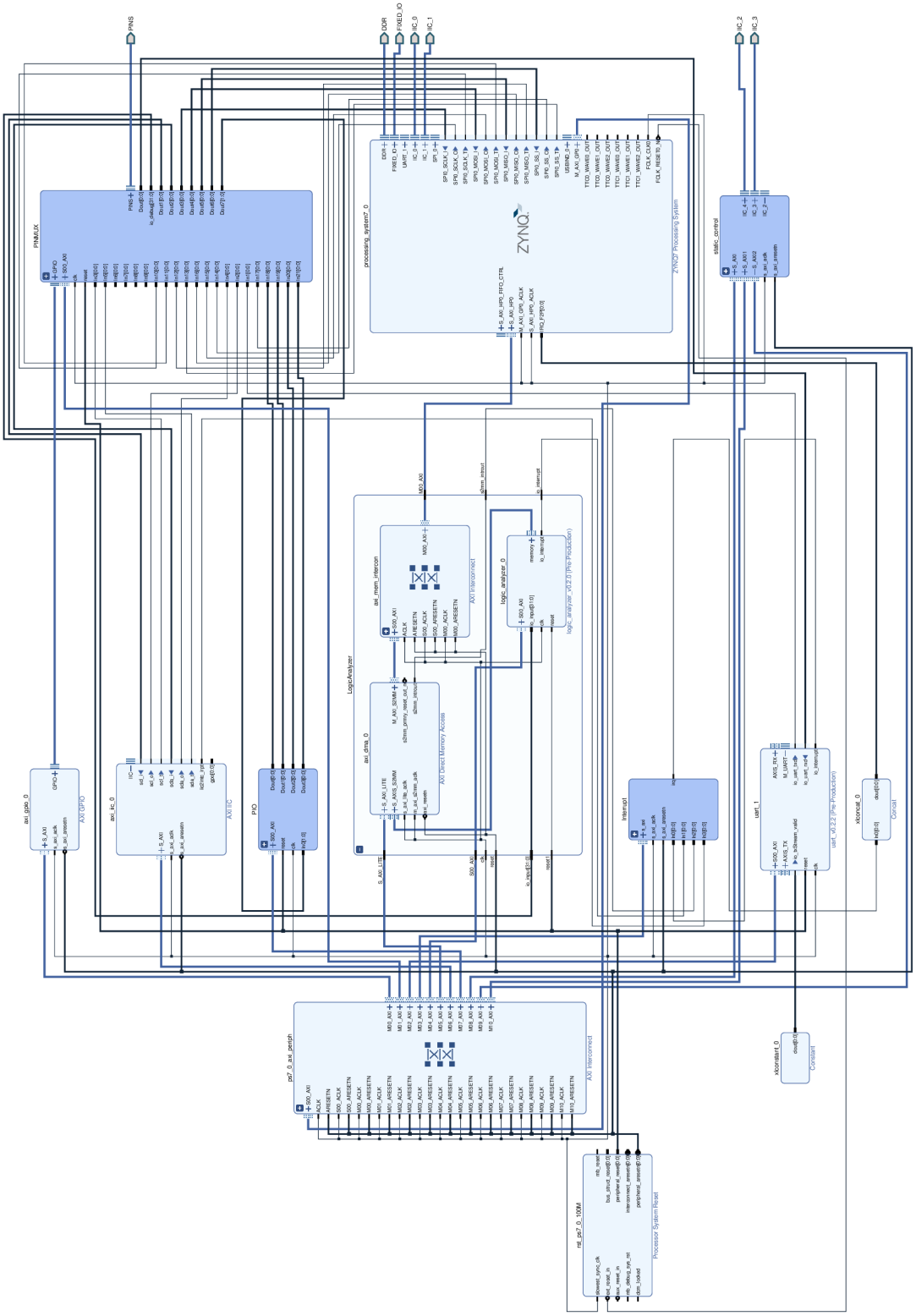
Příloha B

Schéma adaptéru


Does not have back power protection!
J1
Raspberry_Pi_2_3
A1
Arduino_Nano
A2
Arduino_UNO_R3
U1
ESP32-DevKitC
U2
SC0915
3V3_EN – onboard PSU enable.
Automatically enabled via pullup
from usb port.
RUN – internally pulled up.
1 – RESET, 0 – RUN
Test points
not accessible
via pinheader
J2
PIN A
J3
PIN B
J4
I2C_1
J5
VCC_SELECT
J6
I2C_0
Dominik Müller
Sheet: /
File: universal_board.kicad_sch
Title: Universal Adapter Board
Size: A4
Date: 2022-05-13
Rev:
KiCad E.D.A. kicad (6.0.0)
Id: 1/1

Příloha C

Blokové schéma návrhu uvnitř FPGA


PNS
DDR
FIXED_IO
IIC_0
IIC_1
IIC_2
IIC_3
PNMUX
processing_system7_0
ZYNQ7 Processing System
static_control
axi_gpio_0
AXI GPIO
axi_iic_0
AXI IIC
PIO
axi_dma_0
AXI Direct Memory Access
axi_mem_intercon
AXI Interconnect
logic_analyzer_0
logic_analyzer v0.2.0 (Pre-Production)
Interrupt
uart_1
uart_v0.2.2 (Pre-Production)
xlconcat_0
Concat
xlconstant_0
Constant
ps7_0_axi_periph
AXI Interconnect
rst_ps7_0_100M
Processor System Reset