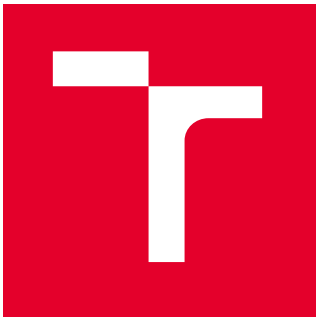


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

TESTBED PRO SIMULACI MCU APLIKACE V RTL PROSTŘEDÍ

TESTBED FOR SIMULATION OF MCU APPLICATION USING RTL ENVIRONMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Petr Ohnút**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Arm, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Petr Ohnút ***ID:*** 203310

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Testbed pro simulaci MCU aplikace v RTL prostředí

POKYNY PRO VYPRACOVÁNÍ:

Úkolem je rozšířit stávající testbed (na bázi test-bench) pro MCU simulaci embedded aplikace v prostředí FPGA, resp. RTL. Testovací nástroj automatizovaně provede simulovaný běh aplikace dle zadaného předpisu (scénáře), přičemž vytvoří podrobný výstup z chodu.

1. Seznamte se stávajícím stavem testbedu.
2. Navrhněte rozšiřující úpravy pro nové funkce a automatizaci procesu.
3. Navrhněte a vytvořte testovací scénáře (testovací sadu).
4. Implementujte navrhované funkce do testbedu.
5. Proveďte testování, pokuste se zpracovat výstupní data a vyhodnoťte výsledky.

DOPORUČENÁ LITERATURA:

Broekman, Bart. Testing Embedded Software. Addison-Wesley, 2003. ISBN: 978-0321159861

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 17.5.2023

Vedoucí práce: Ing. Jakub Arm, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

ABSTRAKT

Diplomová práce je zaměřena na vytvoření testovacího frameworku pro jednoduchou možnost simulování a konfigurování mcu aplikací. Framework také zabezpečuje základní zpracování výstupních dat simulace, jakým je například měření UART či SPI rychlosti komunikace, kontrola očekávané instrukce s aktuálně vykonávanou, počítání vykonaných jednotlivých funkcí během simulace, etc. V rámci práce jsou navrženy testovací scénáře, které mají za úkol simulovat implementované funkcionality frameworku. V poslední řadě jsou rozebírány výsledky jednotlivých testovacích scénářů.

KLÍČOVÁ SLOVA

Soft-core, Vivado, Simulace, RISC-V, FreeRTOS, SystemVerilog, NEORV32

ABSTRACT

The thesis is focused on creating a test framework for easy simulation and configuration of mcu applications. The framework also provides basic processing of simulation output data, such as measuring UART or SPI communication speed, checking the expected instruction with the currently executed one, counting the executed individual instructions during the simulation, etc. Test scenarios are designed to simulate the implemented functionalities of the framework. Finally, the results of each test scenario are discussed.

KEYWORDS

Soft-core, Vivado, Simulation, RISC-V, FreeRTOS, SystemVerilog, NEORV32

OHNÚT, Petr. *Testbed pro simulaci MCU aplikace v RTL prostředí*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2023, 79 s. Diplomová práce. Vedoucí práce: Ing. Jakub Arm, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Petr Ohnút
VUT ID autora:	203310
Typ práce:	Diplomová práce
Akademický rok:	2022/2023
Téma závěrečné práce:	Testbed pro simulaci MCU aplikace v RTL prostředí

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno 17.5.2023 .....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Jakubu Armovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Diplomová práce se zabývá implementací automatického testovacího frameworku mcu aplikací, který z konfiguračního souboru načítá testovací sadu aplikací, jež je následně simulována. Dochází i na zpracování výstupních dat jednotlivých testovacích scénářů. K simulaci aplikací se využívá simulačního prostředí Vivado a soft-core RISC-V procesoru NEORV32.

Práce navazuje na práci [1] a rozšiřuje ji o celkové automatizování procesu, jednodušší konfiguraci testovacích scénářů, nové testovací scénáře a o zpracovávání výstupních dat simulace.

V první polovině práce jsme v krátkosti seznámeni s architekturou RISC-V, samotným testovaným procesorem NEORV32 a s operačními systémy RTOS. Dále jsme seznámeni s prací [1] a jsou navrženy rozšiřující úpravy společně s novými testovacími scénáři. Při návrhu testovacích scénářů je kladen důraz převážně na FreeRTOS a na simulaci nežádoucích stavů, které mohou nastat v těchto aplikacích. Pod nežádoucími stavy si můžeme představit například deadlock, či race condition.

V druhé polovině práce jsme seznámeni se samotnou implementací a principem funkcionality testovacího frameworku. Je vysvětleno, jakým způsobem dochází ke zpracovávání, a vyhodnocování dat a jsou představeny nové testovací scénáře.

V poslední části jsou diskutovány výsledky získané simulacemi pro jednotlivé testovací scénáře. Jsou vyhodnoceny komunikační rychlosti UART a SPI, a je demonstrováno, jakým způsobem by za pomocí získaných dat ze simulace mohl být odhalen nežádoucí stav, který nastane ve FreeRTOS aplikaci.

1 RISC-V

RISC-V je volně dostupná architektura instrukční sady, dále jen ISA (instruction-set architecture), z rodiny RISC, jejíž cílem je podpořit výzkum a vzdělání v oblasti počítačové architektury. V posledních letech se RISC-V architektura stává standardem v oblasti průmyslového využití. Mezi hlavní cíle RISC-V můžeme zařadit:

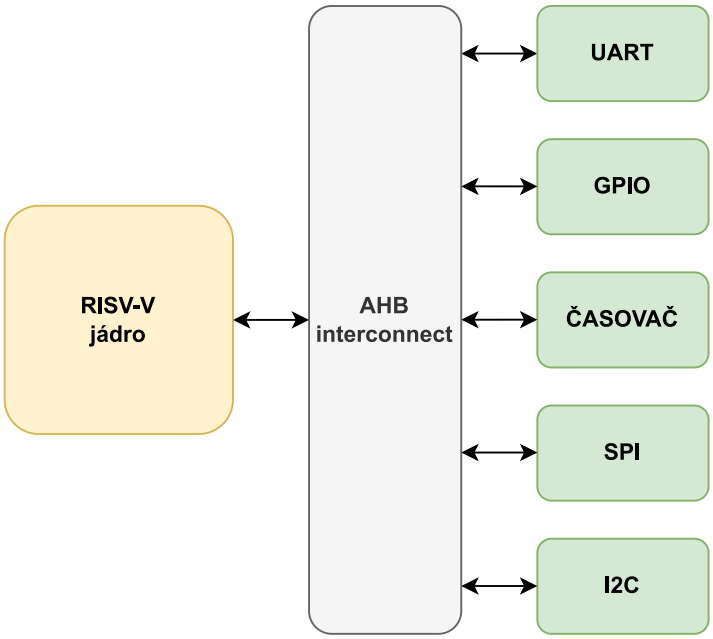
1. Zcela otevřená ISA, která je volně dostupná jak v akademické sféře tak i v průmyslu.
2. ISA vhodnou pro hardwarovou implementaci, nikoli pouze pro simulaci nebo binární implementaci.
3. ISA, která se vyhýbá "nadměrné architektuře"pro určitý styl mikroarchitektury (např. mi-crocoded, in-order, decoupled, out-of-order) nebo implementační technologii (např. ASIC, FPGA), ale která umožňuje efektivní implementaci v kterékoli z nich.
4. Podpora standardu IEEE-754 pro plovoucí desetinnou čárku [2].
5. ISA podporující rozsáhlá rozšíření ISA na uživatelské úrovni.
6. 32bitové i 64bitové varianty adresového prostoru pro aplikace, jádra operačních systémů a další hardwarové implementace.
7. ISA s podporou vysoce paralelních vícejádrových nebo vícejádrových implementací.
8. Volitelné instrukce s proměnlivou délkou, které rozšiřují dostupný prostor pro kódování instrukcí i podporu volitelného hustého kódování instrukcí pro zvýšení výkonu, statickou velikost kódu, a energetické účinnosti.
9. Plně virtualizovatelný ISA pro usnadnění vývoje hypervizoru.
10. ISA, která zjednodušuje experimenty s novými návrhy ISA na úrovni supervizoru a hypervizoru.

Kapitola popisující RISC-V architekturu vychází ze zdrojů [3], [4], [5] a [6].

1.1 Hardwarová platforma

Platformu můžeme chápat jako blok jehož součástí je RISC-V jádro společně se vstupně/výstupními periferiemi, které definují komunikační rozhraní tohoto bloku. RISC-V platforma může obsahovat jedno či více RISC-V kompatibilních CPU jader společně dalšími RISC-V nekompatibilními CPU jádry, fyzickými pamětmi, akcelerátory, vstupně/výstupními bloky a propojujícími prvky, které zajišťují komunikaci mezi zmíněnými komponentami. Zjednodušenou platformu si můžeme prohlédnout na 1.1.

Jádrem myslíme komponentu, jež obsahuje jednotku pro načítání instrukcí. RISC-V kompatibilní jádro může podporovat více hardwarových RISC-V vláken, nebo



Obr. 1.1: Zjednodušená platforma

harts, pomocí multithreadingu. Pojmem hart je myšlen RISC-V výpočetní kontext, který obsahuje plnou sadu RISC-V registrů a který provádí svůj program nezávisle na ostatních hartech v RISC-V systému.

RISC-V jádro můžeš mít dodatečné specializované rozšíření instrukční sady nebo přidaný koprocesor. Koprosecorem myslíme komponentu, která je připojena k RISC-V jádru a je řízená převážně sekvencemi RISC-V instrukcí. Koprocesor obsahuje i další rozšíření instrukční sady a případně i omezenou autonomii vůči primárnímu sekvencím RISC-V instrukcí.

Akcelerátorem je myšlenka ne-programovatelná jednotka, která je určena ke specifické funkci. Může se jednat také o jádro, které může pracovat autonomně, ale je opět specializováno na určitý typ úkolu. V RISC-V platformách se očekává, že akcelerátory budou tvořeny převážně RISC-V jádry se specializovanými instrukčními sadami a/nebo přizpůsobenými koprocesory. Důležitou součástí akcelerátoru je *I/O* akcelerátor, který zajišťuje obsluhu vstupně-výstupních úloh z hlavní aplikace.

RISC-V hardwarová platforma může obsahovat od jedno-jádrového mikrokontroleru až po po cluster tisíců mnoho-jádrových serverových uzlů se sdílenou pamětí.

1.2 RISC-V softwarové prostředí a harty

Chování RISC-V programu závisí na prostředí, ve kterém jsou tyto programy spouštěny. RISC-V rozhraní spouštěcího prostředí, dále jen EEI (execution enviroment

interface), definuje počáteční stav programu, počet a typ hartů, dostupnost a atributy paměti I/O regionů a obsluhu přerušení či výjimek. Mezi příklady EEI můžeme zařadit například binární aplikační rozhraní Linuxu (Linux application binary interface) nebo binární rozhraní pro nadřazený režim RISC-V (SBI). Implementace RISC-V spouštěcího prostředí může být čistě hardwarová, softwarová nebo kombinací jak softwaru, tak hardwaru. Ku příkladu, opcode pasti a softwarové emulace můžou být využity k implementaci funkcionalit, které nejsou poskytovány hardwarovou implementací. Mezi příklady spouštěcího prostředí můžeme zařadit například:

- "Bare metal"hardwarovou platformu, kde jednotlivé harty jsou implementovaný pomocí fyzických procesorových jader a instrukce tak mají plný přístup do fyzického adresového prostoru. Hardware platforma definuje prostředí pro spuštění, které začíná v okamžiku zapnutí zařízení.
- RISC-V operační systém, který poskytuje více uživatelských prostředí pro spuštění, tím, že sdílí uživatelské harty mezi dostupnými fyzickými vlákny procesoru a řídí přístup k paměti pomocí virtuální paměti.
- RISC-V hypervisory, jež umožňují vícero správcovských prostředí pro spuštění pro hostované operační systémy.
- RISC-V emulátor, jakým je například Spike, QEMU nebo rv8, které emulují RISC-V harty na základním x86 systému a mohou poskytovat uživatelské nebo správcovské prostředí pro spuštění RISC-V programů.

Z pohledu softwaru, který je spouštěn v určeném spouštěcím prostředí, hart je zdrojem, který autonomně načítá a vykonává RISC-V instrukce v daném spouštěcím prostředí. Hart se tedy chová jako hardwarové vlákno i přesto, že v reálném hardwaru je hart časově multiplexován spouštěcím prostředím. Některé EEI podporují vytvoření a mazání dalších doplňujících hartů.

Spouštěcího prostředí je zodpovědné za zajištění postupu každého z hartů v daném spouštěcím prostředí. Pro daný hart je tato odpovědnost pozastavena v případě, že hart vykonává mechanismus, který explicitně čeká na nějakou událost, například na přerušení. Tato odpovědnost končí, pokud je ukončen i samotný hart. Postup můžeme chápat jako následující události:

- Instrukce byla vykonána a výsledek byl zapsán do registru nebo do paměti.
- Past.
- Jakákoli jiná událost definovaná rozšířením, která představuje postup.

1.3 Přehled RISC-V ISA

Základní ISA RISC-V je definována jako ISA pro číselné operace, která musí být součástí každé implementace. Tato ISA je velmi podobná ISA prvních RISC proce-

sorů s výjimkou toho, že neobsahuje sloty pro zpoždění větvení[1] a podporuje volitelné kódování instrukcí s proměnnou délkou. Základní instrukční sada je omezena na minimum instrukcí, které jsou dostačující pro kompilátory, assemblery, linkery a operační systémy. Poskytuje se tak vhodná ISA a softwarový toolchain "kostra", kolem které lze vytvořit přizpůsobenou ISA.

RISC-V je ve skutečnosti spíše rodinou souvisejících ISAs, která se v současné době skládá ze čtyř základních ISAs. Každá z těchto základních instrukčních sad je charakterizována šířkou registrů pro celá čísla, korespondující velikostí adresového prostoru a počtem registrů pro celá čísla. Rozlišujeme mezi dvěma variantami instrukčních sad pro celá čísla, a to: RV32I a RV64I. RV32I poskytuje adresový prostor o šířce 32 bitů, mezitímco RV64I poskytuje adresový prostor o šířce 64 bitů. Základní instrukční sada pro celá čísla využívá dvojkový doplněk pro reprezentaci celých čísel se znaménkem.

RISC-V byl navržen tak, aby podporoval rozsáhlá přizpůsobení a specializace. Základní ISA může být rozšířena o jednu nebo více instrukcí, ale základní instrukce nelze měnit. Rozšíření instrukční sady RISC-V dělíme na standardní a nestandardní. Standardní rozšíření by měla být obecně užitečná a neměla by být v rozporu s jinými standardními rozšířeními. Nestandardní rozšíření mohou být vysoce specializovaná nebo mohou být v rozporu s jinými standardními nebo nestandardními rozšířeními. Rozšíření instrukční sady mohou poskytovat mírně odlišnou funkčnost v závislosti na šířce základních celočíselných instrukcí.

Pro podporu vývoje softwaru je definována sada standardních rozšíření, které poskytují celočíselné násobení/dělení, atomické operace a operace s plovoucí řádovou čárkou. Základní celočíselná ISA je označována jako "I"(s předponou RV32 nebo RV64 v závislosti na šířce celočíselných registrů) a obsahuje celočíselné výpočetní instrukce, načítání celočíselných hodnot, ukládání celočíselných hodnot a instrukce pro řízení toku programu. Standardní rozšíření pro celočíselné násobení a dělení je označováno jako "M"a přidává instrukce pro násobení a dělení hodnot uložených v celočíselných registrech. Standardní rozšíření pro atomické instrukce, označované jako "A", přidává instrukce, které atomicky čtou, modifikují a zapisují do paměti pro synchronizaci mezi procesory. Standardní rozšíření pro operace plovoucí řádovou čárku se "single"přesností je označována jako "F", přidává plovoucí řádové číslice se "single "přesností, instrukce pro výpočty a načítání/ukládání hodnot s touto přesností. Standardní rozšíření pro plovoucí řádovou čárku s "double"přesností, označované jako "D", rozšiřuje čísla s řádovou číslicí na přesnost "double"a přidává instrukce pro výpočty a načítání/ukládání číslic s touto přesností. Standardní komprimované instrukční rozšíření "C"poskytuje 16bitové formy běžných instrukcí.

[1]Situace, kdy se instrukce, následující bezprostředně po instrukci skoku, vykonává před samotným skokem.

Nepředpokládá se, že by nová instrukce měla poskytnout významnou výhodu pro všechny typy aplikací, i přes to, že může být velmi prospěšná pro určitou oblast. Snahy o menší energetickou účinnost nutí k větší specializaci a je tedy důležité zjednodušit požadovanou část specifikace ISA. Zatímco jiné architektury obvykle považují své ISA za jednotnou entitu, která se mění na novou verzi s tím, jak jsou postupně přidávány instrukce, RISC-V usiluje o konstantní udržení základního a každého standardního rozšíření a další nové instrukce se stanou součástí dalšího volitelného rozšíření.

1.4 Kódování instrukcí

Základní ISA RISC-V má instrukce pevné délky 32 bitů, které sou zarovnány na 32 bitů. Standardní kódování RISC-V instrukcí je navrženo tak, aby podporovalo rozšíření ISA o instrukce proměnné délky, kde každá instrukce se může skládat z libovolného počtu 16bitových částí, kdy tyto části musí být zarovnány na 16-bitů. Komprimované rozšíření ISA poskytuje 16 bitové instrukce a umožňuje, aby všechny instrukce mohly být zarovnány na libovolnou 16 bitovou hranic.

Termín IALIGN (v bitech) používáme pro označení zarovnání adres instrukcí, které se používají v rámci implementace. V základním ISA je IALIGN 32 bitů, ale některá rozšíření ISA, například komprimované rozšíření, umožňují, aby IALIGN bylo 16 bitů. Hodnota IALIGN vždy musí být 16 nebo 32.

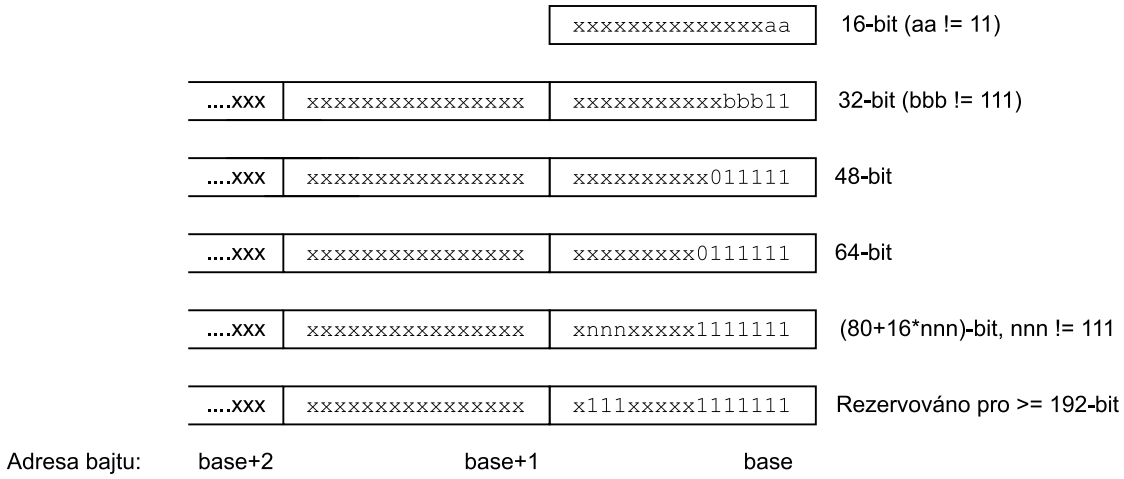
ILEN (v bitech) používáme k označení maximální délky instrukce podporované implementací. ILEN je vždy násobkem IALIGN. Pro implementace podporující pouze základní sadu instrukcí je ILEN 32 bitů.

Obrázek 1.2 znázorňuje kódovací konvenci pro délku RISC-V instrukcí. Všechny 32 bitové instrukce v základní ISA mají své nejnižší dva bity nastaveny na hodnotu 11. Instrukce komprimované instrukční sady sady mají své nejnižší dva bity nastaveny na 00, 01 nebo 10.

1.4.1 Rozšířené zakódování instrukcí

Část prostoru pro kódování instrukcí s 32 bitovou délkou byla vyhrazena pro instrukce delší než 32 bitů.

Rozšíření standardní instrukční sady na více než 32 bitů má další nižší bity nastaveny na 1. Konvence pro délku 48 bitů a 64 bitů si můžeme prohlédnout na obrázku 1.2. Instrukce o délce mezi 80 a 176 bity jsou kódovány pomocí 3bitového pole v bitech [14:12], které udává počet 16bitových slov navíc k prvním 5 $\times$ 16bitovým slovům. Kódování bitů [14:12] na 111 je vyhrazeno pro budoucí kódování delších instrukcí.



Obr. 1.2: Kódování instrukcí v RISC-V.

Kódování bitů [15:0] na nulu je definováno jako neplatné instrukce. Tyto instrukce jsou považovány, že mají minimální délku 16 bitů, pokud je k dispozici rozšíření instrukční sady s délkou 16 bitů, jinak 32 bitů. Kódování bitů [ILEN-1:0] na jedničky je také neplatné; tato instrukce je považována za ILEN bitů dlouhou.

Základní instrukční sady RISC-V mají buďto little-endian nebo big-endian paměti, přičemž privilegovaná architektura definuje big-endian operace. Instrukce jsou v paměti uloženy jako posloupnost 16bitových little-endian parcel, bez ohledu na endianitu paměti. Parcely tvořící jednu instrukci jsou uloženy na násobcích adres půlslova (16 bitů), kdy parcela s nejnižší adresou obsahuje nejnižší bity instrukce.

2 NEORV32

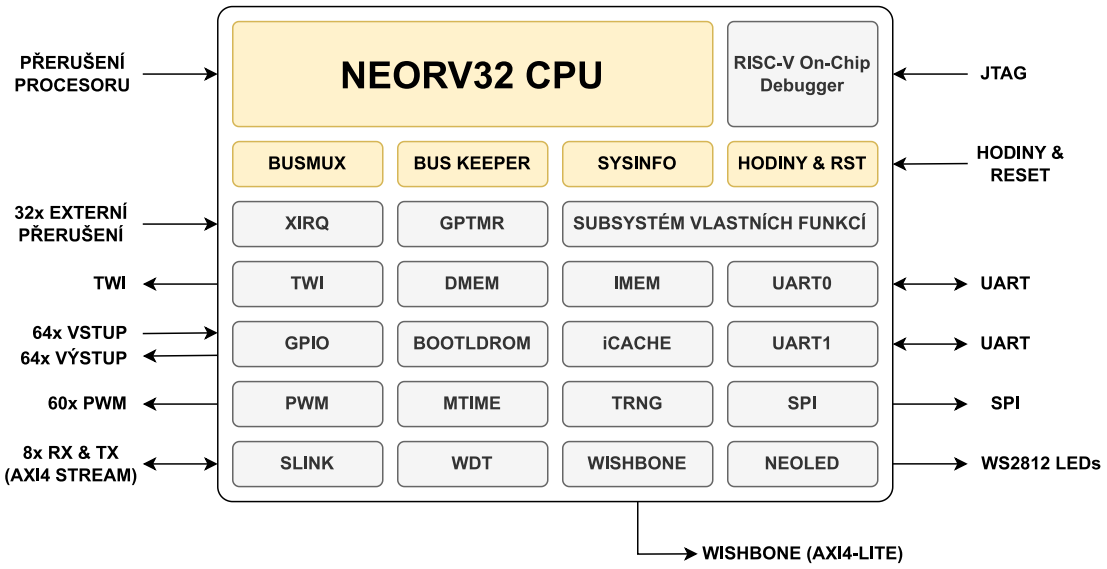
Procesor NEORV32 RISC-V je open-source RISC-V procesorový systém, který je určen jako pomocný procesor v rámci větších návrhů SoC nebo jako samostatný mikrokontrolér.

Systém je vysoce konfigurovatelný a umožňuje uživateli použít běžné periferie, jakými jsou například vestavěné paměti, časovače, sériová rozhraní atd. Systém je možno ladit pomocí debuggeru kompatibilním s OpenOCD/gbd, jež je přístupný přes JTAG.

Pozornost je věnována i bezpečnosti, a to do takové míry, aby byla zajištěno definované a předvídatelné chování v každém okamžiku. Procesor zajišťuje, aby všechny přístupy do paměti byly potvrzeny a aby nebyly prováděny neplatné/chybné instrukce. Kdykoli nastane neočekávaná situace, je aplikační kód informován prostřednictvím hardwarových výjimek. [7]

2.1 System on chip

NEORV32 System on chip spolu s běžnými periferními rozhraními a vestavěnými paměťmi poskytuje plnohodnotnou SoC platformu na bázi RISC-V. Na obrázku 2.1 si můžeme prohlédnout NEORV32 platformu.



Obr. 2.1: NEORV32 platforma

NEROV32 CPU

Samotný procesor je blíže popsána v práci [1].

3 RTOS

V následující podkapitole si přiblížíme a vysvětlíme principy operačních systémů reálného času RTOS a seznámíme se s jeho distribucí FreeRTOS. V rámci diplomové práce jsou navrženy testovací scénáře na základě FreeRTOS. Tyto scénáře slouží k simulaci nežádoucích stavů, které v RTOS systémech mohou nastat.

Kapitola vychází z následujících zdrojů : [8], [9], [10], [11], [12], [13], [14] a [15]

3.1 Charakteristika

Operační systémy reálného času neboli RTOS (**R**eal **T**ime **O**perating **S**ystem), jsou operační systémy, které mají dvě hlavní vlastnosti, a to: determinismus a předvídatelnost. V RTOS systémech se úkoly vykonávají v určitých pevně daných časových intervalech, mezitím, co v klasických operačních systémech tohle není pravidlem. Předvídatelnost a determinismus jdou v tomto případě ruku v ruce: Víme, jak dlouho bude úkon trvat a víme jak se systém během toho bude chovat.

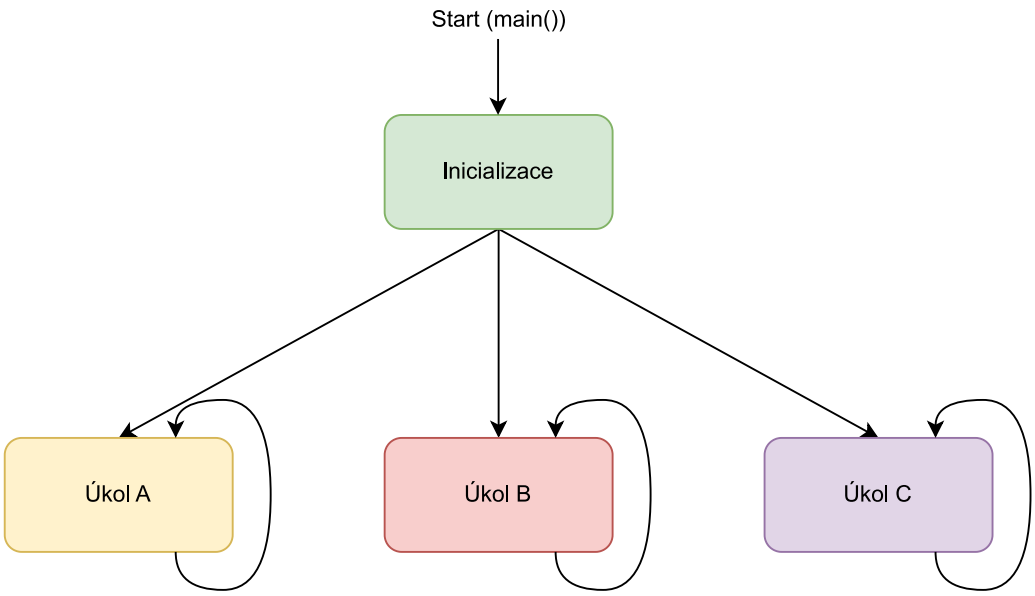
Za hlavní charakteristiky RTOS můžeme považovat:

- **Determinismus**: Systém dokáže zajistit, že každá úloha bude vykonávána v přesně daný časový okamžik a s předem definovanou rychlostí.
- **Vysoký výkon**: Systémy RTOS jsou rychlé a pohotové, často provádějí akce během malého zlomku času než který by potřeboval obyčejný operační systém.
- **Bezpečnost**: Systémy RTOS se často používají v kritických systémech, jejichž selhání může mít katastrofální následky, například v robotice nebo v řídicích systémech letadel. Aby ochránily své okolí, musí mít vysoké bezpečnostní standardy a velmi spolehlivé bezpečnostní funkce.
- **Prioritní plánování**: Prioritní plánování znamená, že akce s vysokou prioritou jsou prováděny jako první a akce s nižší prioritou jsou prováděny až po nich. To znamená, že RTOS vždy jako první provede nejdůležitější úlohu.
- **Malé požadavky na paměť**: Oproti obyčejným operačním systémům, systémy RTOS zabýrají pouze zlomek paměti. Ku příkladu, operační systém Windows 10, se všemi aktualizacemi, zabýrá přibližně 20GB, mezitím co, RTOS VxWorks[1] je přibložně 20 000-krát menší.

Princip jednoduché RTOS aplikace si můžeme prohlédnout na obrázku 3.1.

RTOS operační systémy dále dělíme na dva typy a to : "soft"RTOS systémy a "hard"RTOS systémy.

[1] `https://www.windriver.com/products/vxworks`



Obr. 3.1: Princip RTOS aplikace

3.1.1 Soft RTOS systémy

Soft RTOS systémy mají většinou větší velikosti souborů než soubory používané v hard RTOS systémech. Při vysokém vytížení se soft RTOS systémy mohou někdy chovat poněkud nepředvídatelně, ale tohle je tolerováno, protože v případě vyskytnutí chyby se program vrací k dříve stanovenému kontrolnímu bodu. V soft RTOS systémech je dále povolena drobná časová odchylka k vykonávaným operacím. Je také možné, že systém bude pokračovat v běhu i v případně vyskytnutí chyby, některé funkce ovšem nemusejí pracovat správně.

Soft RTOS systémy se používají převážně v aplikacích, kdy časování nehraje velkou roli. Může se jednat například o osobní počítače, foťáky, případně třeba i chytré telefony.

3.1.2 Hard RTOS systémy

Hard RTOS systémy mají menší datové soubory a pokud dojde k chybě, tak se systém chová přesně podle očekávání a program se vrátí zpět, a to i během velmi vysokého zatížení systému. Hard RTOS systémy se používají převážně v aplikacích, kdy časování je kriticky důležité, může se jednat například o systémy autopilotů, lékařské přístroje atd.

V hard RTOS systémech je časování velmi důležité a nedodržení termínu může mít vážné následky, například zranění člověka nebo poškození zařízení.

3.2 Architektura

Systém RTOS obvykle používá buďto monolitickou nebo mikrojádrovou architekturu.

3.2.1 Monolitická

V monolitické architektuře jádro systému (kernel) obsahuje všechny funkce potřebné pro běh aplikací a správu prostředků. Všechny služby a ovladače jsou tedy implementovány v jádře RTOS a aplikace běží přímo nad jádrem. RTOS monolitické architektury má obvykle menší režii a menší nároky na paměť, ale zároveň je méně flexibilní a méně modulární než RTOS s mikrojádrem. Pokud je potřeba změnit nebo vylepšit jádro, musí se znovu přeložit celý RTOS a je potřeba znovu nahrát novou verzi na cílové zařízení.

Monolitickou architekturu můžeme shrnout do následujících bodů:

- Jádro je srdcem operačního systému, poskytuje základní služby ostatním komponentám a slouží jako hlavní vrstva mezi operačním systémem a hardwarem.
- Monolitické jádro a operační proces sdílejí jeden prostor což umožňuje vyšší výkon ve srovnání s mikrojádrovými konfiguracemi.
- Monolitické systémy jsou rychlé, ale obtížněji se aktualizují a programové chyby v souborovém systému, zásobníku protokolů nebo ovladačích mohou způsobit pád systému.

3.2.2 Mikrojádrová

Mikrojádrová architektura RTOS se liší od monolitické architektury tím, že jádro systému obsahuje pouze základní funkce jako plánovač procesů, správu paměti a komunikaci mezi procesy. Všechny další služby, jako jsou síťové funkce nebo ovladače periférií, jsou implementovány jako samostatné moduly, které běží mimo jádro. Tato architektura má několik výhod oproti monolitické architektuře, jako je vyšší flexibilita a modulárnost. Díky tomu lze snadno měnit nebo vylepšovat pouze konkrétní moduly, aniž by bylo nutné znovu překompilovat celý operační systém. To umožňuje snazší údržbu a vylepšování systému a také zmenšení režie jádra.

Na druhou stranu má mikrojádrová architektura také své nevýhody. Implementace služeb jako samostatných modulů mimo jádro může vést k vyšší režii, většímu využití paměti, komunikace mezi moduly nemusí být tak efektivní a tato implementace může být náchylnější na chyby při vývoji.

Mikrojádrovou architekturu můžeme shrnout následovně:

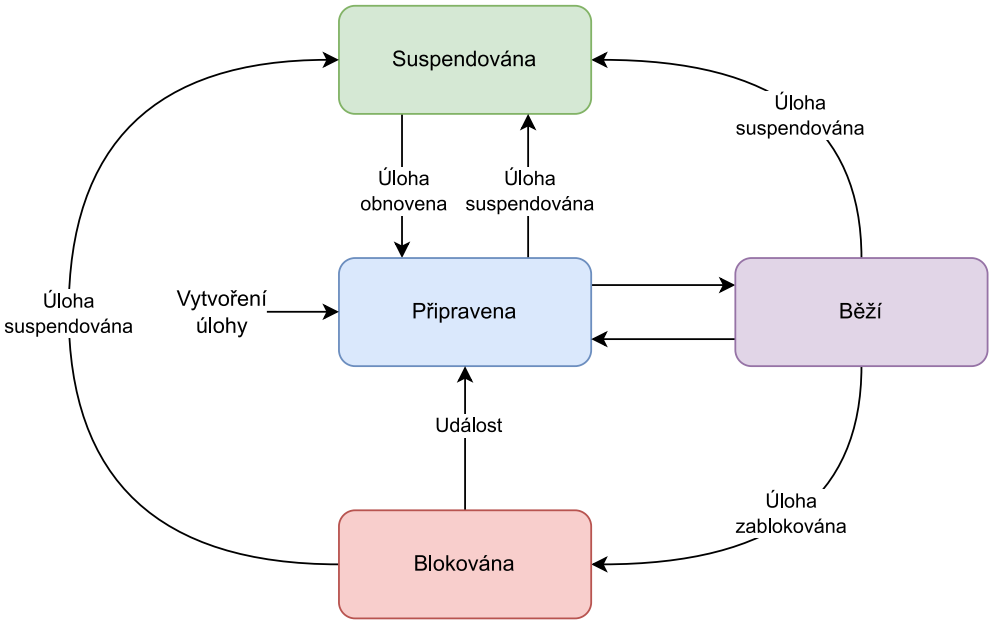
- Jádro a operační procesy se nacházejí na oddělených místech. Tato architektura je pomalejší než monolitická a to z důvodu, že všechny operace se musí vrátit do jádra dříve, než jsou předány komponentě, na kterou se odkazují.
- Mikrojádrová architektura nemá souborový systém.
- Mikrojádrové systémy se jednodušeji programují, aktualizují a jsou odolnější vůči programátorským chybám.

3.3 Plánovač

Typicky úlohy v RTOS systémech mají tři stavy, a to:

- Běží (vykonává se na CPU).
- Připravena být vykonána.
- Blokována (čeká na událost, například na I/O).

Pro lepší představu jak jednotlivá úloha může přecházet mezi jednotlivými stavy se můžeme podívat na obrázek 3.3.



Obr. 3.2: Přechod úlohy mezi stavy

Většinu času je většina úloh buďto připravena, nebo blokována, protože obecně může na jednojádrovém CPU běžet pouze jedna úloha. Počet úloh ve frontě, které jsou připraveny k vykonání se může výrazně lišit v závislosti na počtu úloh, které musí systém vykonat a typu plánovače, který systém používá. U jednodušších nepreemptivních, ale stále víceúlohových systémů musí úloha uvolnit CPU ostatním úlohám, což může způsobit, že vysoký počet úloh je ve frontě a čeká na vykonání, což může vést k vyhladovění.

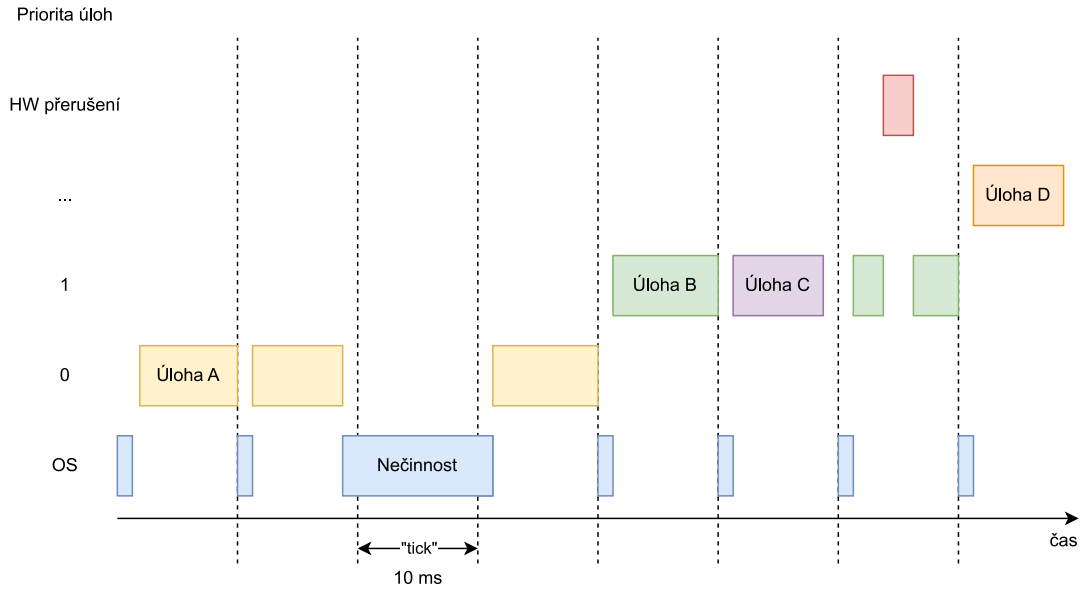
Obvykle je datová struktura fronty připravených úloh v plánovači navržena tak, aby minimalizovala dobu strávenou v kritické sekci plánovače, během které je zabráněna preempce[2] a v některých případech jsou zakázána veškerá přerušení. Volba datové struktury závisí také na maximálním počtu úloh, které mohou být v připravené frontě.

Pokud máme jistotu, že ve frontě se objeví pouze několik úloh, používá se například dvousměrně vázaný seznam pro seznam připravených úloh. Při možnosti, že úloh již bude více, měla by fronta brát v potaz i seřazení úloh podle priority. Tím bude zajištěno, že nalezení úlohy s nejvyšší prioritou nebude vyžadovat procházení celého seznamu.

Je třeba ovšem brát mít na paměti, aby během prohledávání fronty nedošlo například k přerušení a tento proces nebyl pozastaven. Delší kritické sekce by měly být rozděleny na menší části. Pokud nastane přerušení, které učiní vysokoprioritní úlohu připravenou během vkládání nízkoprioritní úlohy do fronty, tato vysokoprioritní úloha může být vložena a spuštěna ihned před vkládáním nízkoprioritní úlohy.

Kritická doba odezvy, někdy nazývaná doba návratu, je doba, která je potřeba na zařazení nové připravené úlohy a obnovení stavu nejvyšší-prioritní připravené úlohy pro spuštění. V dobře navrženém RTOS by připravení nové úlohy mělo trvat 3 až 20 instrukcí na každou položku ve frontě připravených úloh a obnovení nejvyšší-prioritní připravené úlohy by mělo trvat 5 až 30 instrukcí.

Průběh RTOS apliakce společně s demonstrací priorit jednotlivých úloh si můžeme prohlédnout na obrázku 3.3.



Obr. 3.3: Průběh RTOS aplikace

[2]Schopnost operačního systému přerušit běžící proces a přepnout na jiný proces s vyšší prioritou.

Mezi běžně používané algoritmy pro plánovače patří například kooperativní multitasking nebo preemptivní plánování.

3.3.1 Kooperativní multitasking

V kooperativním multitaskingu, záleží na jednotlivých úlohách, zda si během svého běhu povolí přerušení a uvolní tak CPU pro jinou úlohu. Tento přístup vyžaduje určitou formu spolupráce mezi jednotlivými úlohami, protože každá úloha musí CPU uvolnit. Jestliže některé z úloh CPU neuvolní, tak může dojít k zablokování celkového systému.

Úloha v kooperativním multitaskingu může běžet neomezeně dlouho, dokud sama neuvolní CPU. Je tedy nutné, aby programátor zaručil, že každá úloha řádně uvolní CPU, aby bylo zajištěno, že i jiné úlohy budou moci být spuštěny. Tento způsob může být výhodný v případech, kdy jsou na sobě úlohy úzce závislé a vyžadují úzkou spolupráci. Tento přístup ovšem může vést i k problémem s nestabilitou systému a to v případech, kdy úloha neuvolní CPU nebo pokud se úloha vykonává příliš dlouho a neuvolní tak CPU pro úlohy další.

3.3.2 Preemptivní plánování

Preemptivní plánování (také nazývané plánování s přerušením) je typ plánování procesů, které umožňuje jádru operačního systému přerušit běžící proces a nahradit ho jiným procesem s vyšší prioritou, pokud se naskytne nutnost. To znamená, že jádro může odebrat procesoru kontrolu nad běžícím procesem a přidělit ji jinému procesu, který má vyšší prioritu, aby se urychlilo zpracování kritických úkolů a minimalizovalo se zdržení. Preemptivní plánování je běžně používáno v reálném čase operačních systémech (RTOS) a zajišťuje předvídatelnost a determinismus v čase odezvy na události.

3.4 Komunikace mezi úlohami a sdílení prostředků

RTOS systémy musí zajistit schopnost sdílení dat a hardwarových prostředků mezi několika úlohami. Obvykle není bezpečné, aby dvě úlohy přistupovaly ve stejném čase ke stejným datům a nebo hardwarovým prostředkům. Obvyklým řešením tohoto problému je dočasné maskování/zakázání přerušení, použití mutexů či předávání zprávy mezi jednotlivými úlohami. Tyto metody jsou blíže popsány níže.

3.4.1 Dočasné maskování/zakázaní přerušení

V jednojádrových systémech nejjednodušší způsob, jak zabránit současnému přístupu ke sdíleným prostředkům je maskování přerušení. Mezi sdílené prostředky můžeme zařadit například globální proměnnou programu. Dokud jsou přerušení maskována a aktuálně běžící úloha neprovádí blokující volání, má daná úloha výhradní právo používat procesor, jelikož žádná jiná úloha ani přerušení nemůže převzít řízení, což znamená, že kritická sekce je chráněna. Pokud úloha opustí kritickou sekci, přerušení jsou odmaskována a případná čekající přerušení se vykonají. K dočasnému maskování přerušení by mělo dojít pouze tehdy, pokud je nejdelší cesta přes kritickou sekci kratší než je maximální latence přerušení. Tento způsob ochrany se používá v případech, kdy kritická sekce obsahuje pouze pár instrukcí a neobsahuje žádné smyčky. Tato metoda je ideální například pro ochranu hardwarových bitově mapovaných registrů.

3.4.2 Mutexy

Pokud je potřeba zarezervovat sdílený prostředek, tak, aby nedošlo k zablokování všech ostatních úloh, je výhodné použít mechanismus, které jsou dostupné i v normálních operačních systémech a tím je mutex.

Mutex umožňuje pouze jedné úloze a nebo jednomu vláknu přistupovat ke sdílenému prostředku. Pokud je mutex uzamčený (locked), ostatní procesy či vlákna musí čekat, dokud není mutex odemčený (unlocked). To zajišťuje, že se různé úlohy nebudou vzájemně překrývat a nebudou mít nekonzistentní přístup ke sdíleným prostředkům. Úloha může nastavit časový limit pro čekání na mutex. U návrhů založených na mutexech existuje několik známých problémů jakým je deadlock a nebo inverze priorit.

V případě deadlocku dvě nebo více úloh uzamknou mutex bez časového limitu a pak věčně čekají na mutex druhé úlohy, čímž vzniká cyklická závislost a dochází tedy k situaci známé jako deadlock. Nejjednodušší scénář deadlocku je, když dvě úlohy střídavě zamykají dva mutexy, ale v opačném pořadí. Deadlocku se předejít pečlivým návrhem programu.

Inverze priorit může nastat, pokud má úloha s vyšší prioritou závislost na prostředcích, které jsou využívány blokující úlohou s prioritou nižší. Blokující úloha může být v době svého běhu přerušena úlohou s vyšší prioritou a ta může čekat na uvolnění prostředků, které drží blokující úloha.

3.4.3 Předávání zprávy

Dalším způsobem, jak přistupovat ke sdílení prostředků je posíláním zpráv řídící úloze. V této metodě je sdílené prostředek spravován pouze jednou úlohou a ostatní úlohy, které chtějí dotazovat nebo obsluhovat tento sdílený prostředek pošlou zprávu spravující úloze. Tato spravující úloha následně vykoná zpracuje požadavek. Jednoduché systémy založené na posílání zpráv se většinou vyhýbají nebezpečí zablokování protokolů a obecně se chovají lépe než mutexové systémy. Mohou se však vyskytnout podobné chyby jako u mutexových metod. Může nastat například inverze priority.

3.5 FreeRTOS

V rámci diplomové práce některé z testovacích scénářů, které jsou blíže popsány v dalších kapitolách, využívají RTOS distribuci FreeRTOS a z toho důvodu si ho krátce představíme.

FreeRTOS je vlastněn, vyvíjen a udržován společností Real Time Engineers Ltd a je ideální pro vestavěné aplikace reálného času, které využívají mikrokontroléry nebo malé mikroprocesory.

Mezi standardně podporované funkce ve FreeRTOS můžeme zařadit:

Preventivní nebo kooperativní operace

Blíže popsáno v 3.3.

Flexibilní přiřazování priority úlohám

Funkce, která umožňuje dynamicky upravovat prioritu úloh během běhu aplikace.

Flexibilní a rychlý mechanismus oznamování úloh

Tento mechanismus umožňuje jednotlivých úlohám mezi sebou komunikovat a tím synchronizovat akce.

Fronty

Fronty se využívají ke komunikaci mezi úlohami a umožňuje ukládat a odepírat prvky z front typu FIFO.

Binární semafory

Binární semafor je podobně jako mutex synchronizační mechanismus pomocí kterého se řídí přístup ke sdíleným prostředkům. Binární semafor má pouze dva stavy a to volno a obsazeno.

Čítací semafory

Čítací semafor umožňuje, na rozdíl od binárního semaforu, n-krát vstoupit ke sdíleným prostředkům. Pokud tento sdílený prostředek využívá již n úloh, ostatní musí čekat, dokud se neuvolní místo pro další úlohu.

Mutexy

Blíže popsáno v 3.3.

Rekurzivní mutexy

Rekurzivní mutex umožňuje dané úloze opět získat mutex bez nutnosti ho uvolňovat.

Softwarové časovače

Softwarový časovač umožňuje spouštět úlohy s periodickým intervalem.

Skupiny událostí

Skupiny událostí jsou mechanismem, jenž umožňuje se úlohám synchronizovat a čekat na skupinu událostí. Jedná se tedy o způsob, jak synchronizovat úlohy takovým způsobem, aby se splnily pouze ve chvíli, kdy jsou splněny určité podmínky.

Tick hook funkce

Speciální funkce, která je volána s každým systémovým tickem.

Idle hook funkce

Funkce, která je volána, když nejsou žádné úlohy připraveny k běhu, což umožňuje vykonávat užitečné úkoly, když nejsou žádné úlohy k dispozici.

Kontrola přetečení zásobníku

Kontrola zabezpečuje, že úloha nemá příliš velký zásobník, což by mohlo vést k nestabilitě a následnému pádu systému.

Záznam stopy

Funkce, která umožňuje zaznamenávat činnost a chování FreeRTOS systému.

Shromažďování statistik běhu úlohy

Shromažďování statistik nám umožňuje měřit výkon a monitorovat úlohy.

Volitelné komerční licencování a podpora

FreeRTOS je zdarma k použití a modifikování. Společnost Amazon také poskytuje komerční licenci, která obsahuje dodatečné funkce a podporu pro průmyslové nasazení.

Úplný model vnořování přerušení

Vnořování přerušení umožňuje v případě vzniku přerušení s vyšší prioritou při již běžící obsluze jiného přerušení s nižší prioritou obsloužit prvně přerušení s vyšší prioritou a následně se vrátit k dokonání přerušení s prioritou nižší.

Funkce bez tikání pro aplikace s extrémně nízkou spotřebou energie

Funkce bez tikání umožňuje aplikacím, kde je kladen důraz na extrémně nízkou spotřebu energie pracovat bez tikání systémového časovače.

Softwarově řízený zásobník přerušení

Softwarově řízený zásobník přerušení umožňuje definovat a používat vlastní zásobník přerušení namísto zásobníku, který je využíván jádrem FreeRTOS.

4 Testbed

V následující kapitole se blíže seznámíme s testbedem, který byl navržený v práci [1] a na který diplomová práce navazuje. Dále budou navrženy rozšiřující úpravy, které povedou k celkovému vylepšení využití testbedu.

Cílem práce [1] bylo vytvořit testbed pro simulaci embedded aplikace pro soft-core procesor v prostředí FPGA. Autor zvolil procesor na architektuře RISC-V, a to konkrétně NEORV32. Výsledný testbench umožňuje lépe monitorovat stav celého implementovaného SoC.

4.1 Existující implementace

Jak již bylo zmíněno, tak práce navazuje převážně na [1]. Hlavním prvkem implementace je samotný testbed `neorv32_tb.sv` napsán v jazyce SystemVerilog. Testbed zajišťuje ovládání vstupů samotného testovaného modulu, načítání vstupních dat, který jsou poslána do testovacího modulu a vyčítání změn na vstupu, které jsou zapisovány do výsledného souboru. Dále jsou obsaženy dva skripty v jazyce TCL, kdy první skript `create_project.tcl` zajišťuje vytvoření Vivado projektu a druhý skript `neorv32_test.tcl` spouští samotnou simulaci.

4.1.1 Struktura testbedu

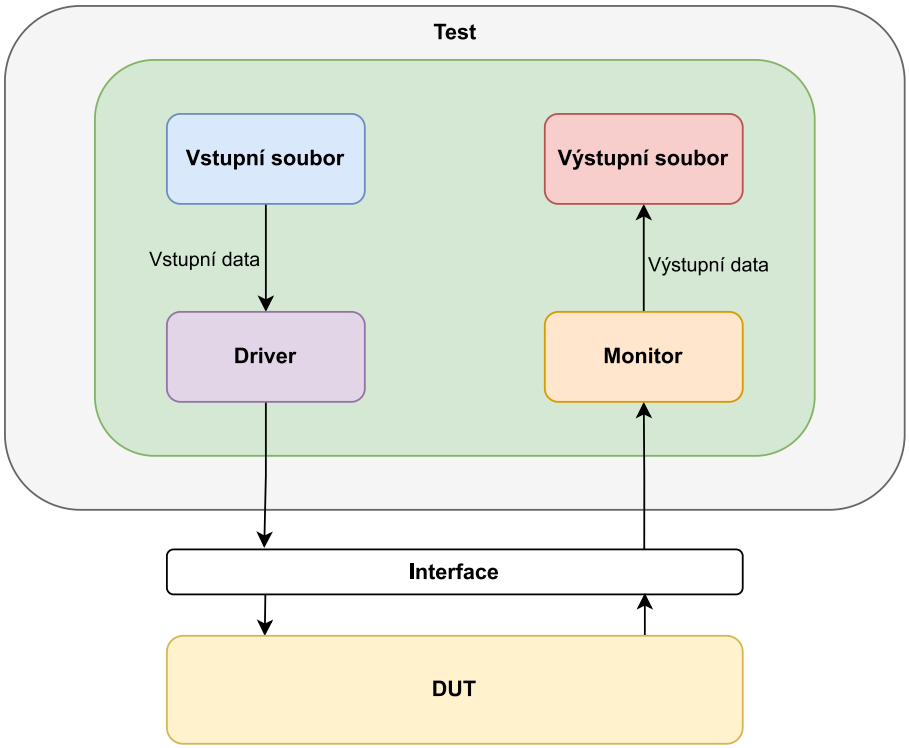
Oproti klasické architektuře testbedů, kdy je obsažen generátor dat a scoreboard, který vyhodnocuje data, tyto dva prvky již navržený testbed neexistuje. Testbed načítá vstupní data z konfigurovatelných vstupních souborům, jež jsou poslána přes driver na interface testované jednotky. Změny na výstupech jsou zaznamenány monitorem a zapsány do výstpného souboru. Struktura testbedu si můžemep prohlédnout na obrázku 4.1.

4.1.2 Použití

Založení projektu a spouštění samotné simulace můžeme shrnout do následujících kroků:

1. Konfigurace testovacího scénáře
2. Spuštění TCL skriptu pro vytvoření Vivado projektu
3. Nastavení paramaterů a spouštění skriptu pro spuštění simulace

Ve výpisu 4.1 si můžeme prohlédnout konfiguraci testovacích scénářů. Jak vidno, tak testovací scénář lze ovlivňovat souborem se vstupními daty, jež jsou poslány přes interface do testované jednotky, délkou simulace a možností zapnout či vypnout PWM.



Obr. 4.1: Struktura testbedu

Výpis 4.1: Příklad konfiguračního souboru z [1]

```
1 . < instruction_code_file.vhd >
2 . < input_file.csv >
3 . < output_file.csv >
4 . < time_of_simulation_in_ns_or_specified_time_unit >
5 . < PWM_filter_enable (0;1) >
```

Samotný princip spuštění simulace si můžeme prohlédnout ve výpisu 4.2. Můžeme vidět, že před samotným spuštěním skriptu je nutné mu nastavit konfigurační soubor jako argument.

Výpis 4.2: Příklad spuštění testovacího skriptu a nastavení jeho parametrů z [1]

```
tcl > cd ../testbench
tcl > set argv ./test_files/config.txt
tcl > set argc 1
tcl > source neorv32_test.tcl
```

4.1.3 Výstupní data

Výstupní data jsou zaznamenávány do výstupního souboru. Formát výstupních dat si můžeme prohlédnout ve výpisu 4.3.

Výpis 4.3: Formát výstupních dat

```
Timestamp[ns];curr_pc_o;gpio_o;PWM_f;PWM_duty;uart_txd_o
```

Ve výpisu 4.3 pak následně :

1. `Timestamp[ns]` je časová stopa
2. `curr_pc_o` je hodnota programového čítače
3. `gpio_o` je hodnota výstupu GPIO
4. `PWM_f` frekvence pwm
5. `PWM_duty` střída pwm
6. `uart_txd_o` je zaznamenaný znak na výstupu UART

Výstupní data nejsou dále zpracovávány.

4.2 Návrh úprav a rozšíření

Hlavní myšlenkou celé úpravy je zautomatizovat celkový proces pomocí skriptu, který zajistí jak vytvoření projektu, nakonfigurování testovacích scénářů, tak i následné spuštění všech či pouze vybraných testovacích scénářů. Cílem by mělo být také vytvořit nové testovací scénáře a pokusit se o zpracování výstupních dat. Návrh dodatečných testovacích scénářů je blíže popsán v kapitole 5.

4.2.1 Návrh testovacího frameworku

Testovací framework by měl nejlépe být pouze jediný skript, napsán například v jazyce Python, kterému se předají vstupní argumenty a následně automaticky vytvoří či načte Vivado projekt, provede konfiguraci testovacích scénář/scénáře, které následně i přeloží, spustí simulaci scénáře/scénářů a následná výstupní data zpracuje.

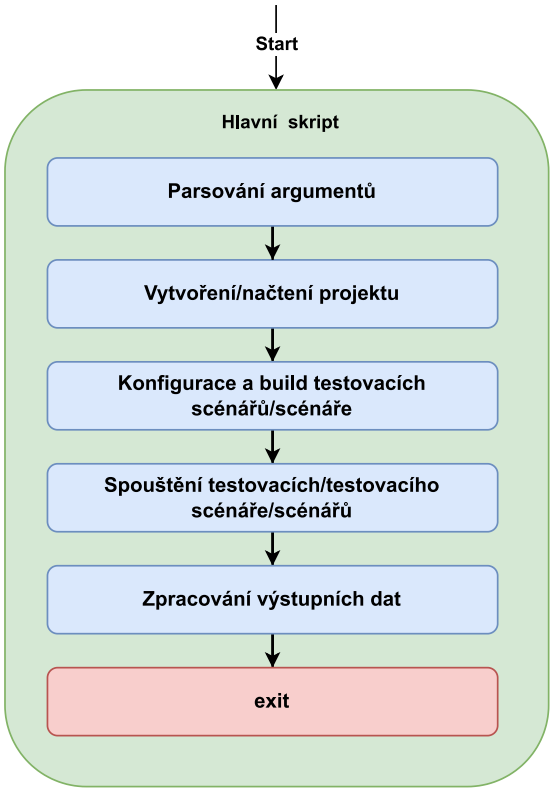
K vytvoření projektu a k spuštění simulace budou využity skripty z [1] a to konkrétně `create_project.tcl` a `neorv32_test.tcl`.

Jednotlivé kroky jsou lépe vyobrazeny v obrázku 4.2.

4.2.2 Rozšíření testbedu a konfigurace scénářů

Cílem je zachovat stávající strukturu testbedu, jak bylo vyobrazeno na obrázku 4.1 a přidat další rozšiřující možnosti ovládání. Aktuální implementace podporuje zaznamenávání programového čítače, GPIO výstupu, frekvence a střídu PWM a výstup UART. Pomocí vstupního souboru je pak možné GPIO a UART ovládat.

Jak sám autor zmiňuje tak dalšími vhodnými kandidáty na rozšíření je například externí přerušení či SPI. V rámci diplomové práce bude testbed rozšířen o SPI a externí přerušení. Dalším vhodným rozšířením je vyvedení signálu, který obsahuje kód právě vykonávané instrukce. Kód instrukce by následně mohl být použit v rámci



Obr. 4.2: Návrh testovacího frameworku

zpracování výsledků a to tím způsobem, že na základě kódu instrukce může být nalezen její ekvivalent v assembler souboru, který je vytvořen při překladu testovacího scénáře, pro lepší čitelnost.

Další možnou úpravou je konfigurovat testovací scénáře, kdy konfigurace jednotlivých scénářů budou obsaženy v souboru formátu `json`. V ideálním případě by bylo vhodné i konfigurovat samotný kód testovacího scénáře a mít možnost upravovat nastavení periferií, například baud rate pro UART. Aby se následně uživateli zamezilo nutnosti takto upravené scénáře překládat ručně, budou scénáře přeloženy taktéž automaticky.

4.2.3 Zpracování výstupních dat

Jak již bylo zmíněno, tak v práci [1] nedochází ke zpracování výsledných dat.

Zpracování výstupních dat bude spočívat k doplnění výstupních dat o assemblerovskou instrukci dle hodnoty kódu právě vykonávané instrukce. Informace o právě vykonávané instrukci nám zaručí lepší čitelnost.

Dalším vhodným zpracováním je vytvoření zaznamenání počtu vykonaných jednotlivých instrukcí a funkcí během běhu simulace a změření rychlosti komunikace UART či SPI.

FreeRTOS testovací scénáře obsahují scénáře, kde záměrně dochází k některým z nežádoucích stavů, například deadlock či race condition. Cílem je demonstrovat, jakým způsobem by nežádoucí stav mohl být odhalen za využití zpracovaných výstupních dat ze simulace.

5 Návrh testovacích scénářů

V následující kapitole si blíže přiblížíme myšlenku jednotlivých scénářů, o které bude set scénářů rozšířen. Nové scénáře bychom mohli rozdělit do dvou skupin, a to scénáře periferií a FreeRTOS scénáře. Jednotlivé scénáře jsou blíže popsány níže.

5.1 Scénáře periferií

Scénáře periferií slouží především k možnosti využívat periferii v rámci testbedu a mít možnost ji ovládat pomocí vstupního konfiguračního souboru a zaznamenávat hodnoty na výstupu této periferie, pokud existují, či k měření komunikačních rychlostí.

5.1.1 SPI

Pro testovací scénář SPI periferie může být, obdobně jak u UART testovacího scénáře z [1], využit loopback. Výstupní port rozhraní SPI je v rámci testbedu připojen ke vstupnímu portu a je tedy možné odeslaná data přijmout a vytisknout pomocí UART. Pro testovací scénář SPI je kladen důraz na rychlost komunikace a vzniklo tedy několik testovacích scénářů, kdy každý scénář má nastavenou jinou komunikační rychlost. Cílem je zaznamenat skutečnou komunikační rychlost rozhraní SPI.

5.1.2 Přerušení

Do testbedu bude implementován driver, který ovládá vstupy externího přerušení do testovaného modulu a navrhnutý scénář by měl pomocí UART vytisknout hodnotu kanálu, ze kterého přijal požadavek na přerušení. Driver nastavuje vstupy na základně vstupního datového souboru.

5.2 FreeRTOS scénáře

Hlavním cílem FreeRTOS scénářů je nasimulovat nežádoucí stavy, ke kterým dochází především v RTOS systémech. Mezi nežádoucí stavy můžeme zařadit například deadlock nebo race condition.

Jednotlivé nežádoucí stavy, které budou simulovány, a jakým způsobem k těmto stavům dochází je blíže popsáno níže.

5.2.1 Deadlock

Deadlockem rozumíme situaci, které nastane v případě, že dvě úlohy na sebe vzájemně čekají, což může vést k zastavení běhu celého systému.

Příkladem deadlocku může být případ, kdy úloha A si vezme mutex 1, spustí úlohu B s vyšší prioritou, která si vezme mutex 2 a následně se snaží vzít mutex 1, který má úloha A. Začne se tedy opět vykonávat úloha A, která po spuštění úlohy B se následně snaží vzít mutex 2 a čeká, dokud si ho nevezme. Úloha A se tedy nikdy nemůže dokonat, jelikož čeká, až se dokončí úloha B, která naopak čeká na dokončení a uvolnění úlohy A, dochází tedy k deadlocku a k zastavení běhu celého systému.

5.2.2 Vyhladovění

K vyhladovění, nebo-li starvation, může dojít v případě, kdy úloze s nižší prioritou není poskytován dostatečný procesorový čas k jejímu vykonávání, protože jsou neustále vykonávány úlohy s prioritou vyšší. V extrémních případech může dojít i k situacím, kdy úloha s nižší prioritou nikdy nedostane prostor k vykonání.

5.2.3 Race condition

Při race condition dochází ke stavu, kdy dvě či více úloh se snaží využívat společný zdroj v různém pořadí či v různých časových intervalech bez toho, aniž by během využívání zdroje jednou úlohou nebyl omezen přístup pro úlohy druhé. Může tak docházet k nekonzistentnímu chování či výsledkům.

K race condition dochází například v situaci, kdy dvě úlohy pracují s globální proměnnou a ukládají do ní výsledek. Úloha A si může proměnnou načíst, provádět výpočty a následný výsledek uložit zpátky, mezitím, co úloha B se může vykonávat rychleji a může například pouze proměnnou inkrementovat. Pokud si úloha A tuto proměnnou načte jako první a během jejího běhu se spustí úloha B, tak tato hodnota, která byla načtena úlohou B, není správná, jelikož úloha A ještě nedokončila svůj výpočet. Může tedy docházet k nekonzistentním a k nepředvídatelným výsledkům.

5.2.4 Unbounded priority inverze

Při nastání unbounded priority inverze úloha s nižší prioritou blokovat úlohu s prioritou vyšší po neomezenou dobu. Toto chování může mít fatální následky a může vést k celkovému selhání systému.

V roce 1997 se uskutečnila mise Mars Pathfinder, která skoro selhala právě na základě této chyby.

6 Implementace navrhovaných změn

V následující kapitole se seznámíme, jakým způsobem byly implementovány navrhované změny a seznámíme se s využitými technologiemi.

6.1 Využité technologie

V následující podkapitole si přiblížíme a popíšeme technologie, které byly využity k realizaci diplomové práce.

6.1.1 Jazyk Python

Jazyk Python patří mezi jedny z nejvíce univerzálních a nejvíce populárních programovacích jazyků. Podporuje moduly, třídy, výjimky, dynamické datové typy na velmi vysoké úrovni a dynamické typování. Má velmi rychlou učící křivku a velmi jasnou syntaxi. Díky své univerzálnosti se využívá například k vývoji webových stránek, datové analýze či k automatizačním úkolům. [16]

V rámci diplomové práce byl Python využit k automatizaci procesu vytvoření Vivado projektu a k automatickému spouštění testovacích scénářů. Byla použita verze `Python 3.10.11`.

6.1.2 WSL

WSL, nebo-li Windows Linux subsystém (Windwos Subystem for Linux), je nástroj, který umožňuje spouštět Linuxové prostředí ve Windows bez nutnosti virtuálního prostředí. WSL se vužívá v rámci automatizovaného frameworku a slouží pro překlad zdrojových kódů testovacích scénářů. WSL byl zvolen pro svou dostupnost a podporu Bash příkazů.

6.1.3 System Verilog

System Verilog patří mezi HDL (**H**ardware **D**escription **L**anguage) jazyky a využívá se pro návrh a simulaci digitálních integrovaných obvodů, například programovatelných hradlových polí nebo různých zákaznických obvodů. SystemVerilog je rozšířen o možnosti použití tříd, které umožňují jednodušší realizaci testovacích scénářů navrženého digitálního designu. [17]

V diplomové práci se SystemVerilog využívá k realizaci testovacích scénářů.

6.1.4 Vivado

Vivado je integrované vývojové prostředí od společnosti Xilinx, který slouží k syntéze, implementaci a analýze HDL designů. Vivado podporuje několik několik funkcionalit, mezi nejdůležitější funkcionality můžeme zařadit například HLS (**H**igh **L**evel **S**ynthesis), která umožňuje vytvářet RTL modely pomocí C, C++ a SystemC programovacích jazyků, Vivado simulátor sloužící k simulaci navržených designů nebo například automatizaci procesů pomocí jazyku TCL. [18]

Jak bylo zmíněno výše, tak v rámci diplomové práce se využívá převážně Vivado simulátor a TCL skripty.

6.1.5 FreeRTOS

FreeRTOS se využívá pro testovací scénáře, a to převážně k nasimulování nežádoucích stavů, které mohou nastat ve FreeRTOS aplikacích.

6.2 Testovací framework

Testovací framework, který, jak bylo zmíněno v 6.1.1, je napsán v jazyce Python slouží ke spuštění simulací všech, či pouze jednoho vybraného testovacího scénáře. Mezi doplňující funkce můžeme zařadit například vygenerování projektu pro vývojové prostředí Vivado, či automatickou konfiguraci a překlad jednotlivých testovacích scénářů.

Jedná se prakticky o zautomatizování diplomové práce [1] společně s rozšiřujícími funkcemi.

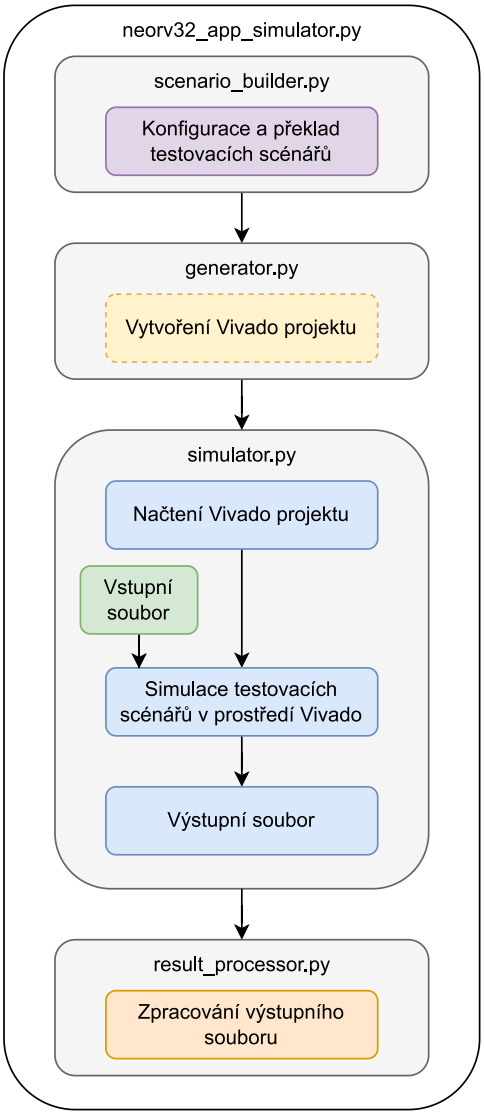
Samotný framework se skládá z několika skriptů, kdy každý skript má svůj specifický úkol. Celkový proces bychom mohli shrnout do čtyř hlavních bodů, a to :

1. Konfigurace a překlad testovacích scénářů
2. Generování Vivado projektu (volitelné)
3. Simulace testovacích scénářů
4. Zpracování výstupních dat

Každý z těchto bodů je zpracováván samostatným skriptem, kdy všechny tyto skripty jsou součástí hlavního skriptu `neorv32_app_simulator.py`, který spouští jednotlivé skripty a slouží k celkové režii všech podskriptů. Jakým stylem je řešen každý ze zvýše zmíněných bodů se dočteme v podkapitolách níže.

Na obrázku 6.1 si můžeme prohlédnout zjednodušenou architekturu testovacího frameworku společně s názvy jednotlivých podskriptů a s jejich úlohou v rámci hlavního skriptu.

Každý skript je řešen jako třída, která má své atributy, které jsou nastavovány během inicializace samotné třídy a nebo během vykonávání metod. Každá třída



Obr. 6.1: Zjednodušená architektura testovacího frameworku

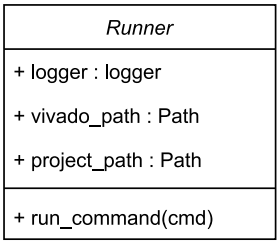
jednotlivého podskriptu je potomkem třídy `Runner` (součástí zdrojového souboru `runner.py`), jež obsahuje obecné metody a atributy využívané všemi potomky. Tato třída je blíže popsána níže.

Pro zachování jednoduchosti a přehlednosti nejsou v diagramech tříd u privátních metod uvedeny vstupní argumenty.

Třída `Runner`

Jak již bylo zmíněno, tak třída `Runner` je rodičovskou třídou pro třídy jednotlivých podskriptů. Mezi atributy této třídy patří logger, jež je nastaven pomocí skriptu `logger.py`, cesta k programu Vivado a cesta k Vivado projektu.

Třída má pouze jedinou metodu a to metodu `run_command`, která zabezpečuje spouštění příkazů v příkazovém řádku. Diagram třídy si můžeme prohlédnout na obrázku 6.2.



Obr. 6.2: Diagram třídy `Runner`

6.3 Konfigurace a překlad testovacích scénářů

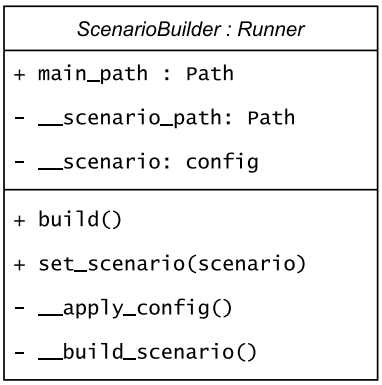
Každý z implementovaných testovacích scénářů může být jednoduše konfigurován a mohou mu být upraveny simulační parametry. Za účel možnosti jednoduché konfigurace byl navrhnout konfigurační soubor `config.json`, který obsahuje konfiguraci každého implementovaného testovacího scénáře. Každý nově implementovaný testovací scénář může být také velmi jednoduše zahrnut do tohoto konfiguračního souboru. Princip konfigurace je blíže popsán níže.

Samotná konfigurace a překlad je řešen pomocí skriptu `scenario_builder.py`, jehož hlavní součástí je třída `ScenarioBuilder`. Jednotlivé metody a atributy třídy jsou popsány dále.

6.3.1 Třída `ScenarioBuilder`

Třída `ScenarioBuilder` je potomkem třídy `Runner` a slouží především ke konfiguraci testovacích scénářů za využití konfiguračního souboru `config.json` a k následd-

nému samotnému překladu zdrojových kódů. Na obrázku 6.3 si můžeme prohlédnout diagram třídy.



Obr. 6.3: Diagram třídy `ScenarioBuilder`

Atributy

Třída obsahuje následující atributy:

`main_path`

Cesta do hlavní složky.

`__scenario_path`

Cesta k aktuálně konfigurovanému a překládanému testovacímu scénáři.

`__scenario_config`

Konfigurace pro aktuální testovací scénář.

Metody

Součástí třídy jsou následující metody:

`build()`

Metoda `build()` je hlavní metodou objektu, která je využita v hlavním skriptu `neorv32_app_simulator.py` a která po zavolání aplikuje konfiguraci, jež obsažena v konfiguračním souboru `config.json`, a přeloží nastavený testovací scénář.

Při překladu je vytvořeny dva soubory nutné k simulaci a k následnému zpracování dat a to aplikační obraz testovacího scénáře `application_image.vhd` a assemblerovský zdrojový soubor testovacího scénáře `main.asm`.

`set_scenario(scenario)`

Metoda slouží k nastavení konfigurace testovacího scénáře.

`__apply_config()`

Konfigurace testovacích scénářů je řešena pomocí generování hlavičkového souboru `config.h` pro každý jednotlivý testovací scénář. Tento soubor je zahrdnut do hlavního zdrojového souboru a obsahuje makra, které se využívají ve zdrojovém kódu a která nastavují například rychlost komunikace UART či rychlost komunikace SPI. Příklad hlavičkového souboru pro testovací scénář `uart_loopback` si můžeme prohlédnout ve výpisu 6.1.

Výpis 6.1: Příklad hlavičkového souboru

```
#define BAUD_RATE 19200
```

`__build_scenario()`

Tato metoda slouží k překladu testovacích scénářů a k vygenerování obrazového souboru aplikace `neorv32_application_image.vhd`, který je nutný ke korektnímu spuštění a vykonání simulace. Jak již bylo zmíněno, tak v případě použití frameworku v operačním systému Windows, využívá se WSL.

6.3.2 Konfigurace testovacích scénářů

Pro jednoduché konfigurování všech testovacích scénářů byl navrhnut soubor ve formátu `json` s názvem `config.json`, který slouží ke konfiguraci všech testovacích scénářů. V rámci tohoto souboru můžeme jednoduše přidávat či odebírat testovací scénáře, nastavovat simulační čas, či konfigurovat například rychlost komunikace UART pro ty scénáře, které UART využívají.

Je možné také nastavit název složky, do které budou uložena výstupní data. Možnost měnit název výstupní složky se hodí v případě, kdy chceme jeden testovací scénář zahrnout do konfiguračního souboru vícekrát a chceme nastavit například jinou komunikační rychlost rozhraní UART.

Speciálně pro FreeRTOS scénáře jsou do configu zahrnouty názvy spouštěných úloh, u kterých je požadováno, aby byl měřen čas mezi opětovným vykonáním úlohy.

Parametrizace byla přidána i do samotného testbedu, kdy je možné nastavovat parametry pro testbed. Tyto parametry jsou následně vloženy do souboru ve formátu `SystemVerilog`, který se přidává do testbedu. Aktuálně je podporováno pouze nastavení rychlosti komunikace UART, která je propagáno do testovaného modulu.

Obecný formát konfiguračního souboru si můžeme prohlédnout ve výpisu 6.2.

Výpis 6.2: Obecný formát konfiguračního souboru

```
{
    "TestScenarios": [
    {
      "name": "<název␣scénáře>",
      "output_folder": "<název_výstupní_složky>",
      "runtime": "<délka␣simulace␣včetně␣jednotky>",
      "pwm" : "<použití␣PWM>",
      "config": [
        "<makra␣pro␣hlavičkový␣konfigurační␣soubor>"
      ]
      "tasks": [
        "<názvy␣úloh>"
      ],
      "config_testbed": [
        "<paramtery␣testbedu>"
      ]
    }
}
```

Název testovacího scénáře musí souhlasit s název složky, ve kterém jsou obsaženy zdrojové soubory. Tato složka také musí být umístěna v předem definované lokalitě a to `<hlavní složka>/testbench/test_scenarios`.

Příklad konfiguračního souboru, který obsahuje nastavení pro dva testovací scénáře, si můžeme prohlédnout ve výpisu 6.3.

Výpis 6.3: Příklad konfiguračního souboru

```
{
    "TestScenarios": [
    {
        "name": "freeRTOS_demo",
        "output_folder": "freeRTOS_demo",
        "runtime": "20ms",
        "pwm" : "0",
        "config": [
            "BAUD_RATE␣19200"
        ],
        "tasks": [
            "Task1",
            "Task2",
```

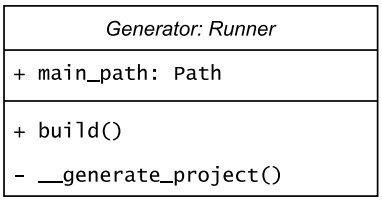
```
                "Task3"
            ],
            "config_testbed": [
                "BAUD_RATE␣=␣19200"
            ]
        },
        {
            "name": "demo_spi",
            "output_folder": "demo_spi_prsc_1",
            "runtime": "5ms",
            "pwm" : "0",
            "config": [
                "BAUD_RATE␣19200",
                "SPI_PRSC␣1"
            ],
            "config_testbed": [
                "BAUD_RATE␣=␣19200"
            ]
        },
    ]
}
```

6.4 Generování Vivado projektu

Generování Vivado projektu je realizováno skriptem `generator.py`, jehož součástí je je třída `Generator`, která obstarává veškerou režii. V rámci generování projektu je využito skriptu `generate_project.tcl` z [1], který je spouštěn prostředím Vivado. Skript byl modifikován tak, aby podporoval vstupní argument `project_dir`, kterým se nastavuje destinace, do které bude projekt vygenerován. Vygenerovaná složka s projektem nese název `vivado`.

6.4.1 Třída `Generator`

Třída `Generator` je potomkem třídy `Runner` a slouží k vygenerování Vivado projektu pomocí skriptu `generate_project.tcl`. Diagram třídy `Generator` si můžeme prohlédnout na obrázku 6.4.



Obr. 6.4: Diagram třídy `Generator`

Atributy

Třída obsahuje následující atributy:

`main_path`

Cesta do hlavní složky.

Metody

Součástí třídy jsou následující metody:

`build()`

Metoda `build()` je hlavní metodou objektu, která je využita v hlavním skriptu `neorv32_app_simulator.py`. Po zavolání této metody dojde k vygenerování Vivado projektu v požadované složce. Složka s projektem nese název `vivado`.

`__generate_project()`

Tato metoda realizuje samotné generování projekt. Projekt je vygenerován pomocí skriptu `generate_project.tcl`, který je spuštěn v prostředí Vivado. Ke spouštění Vivada a k předání vstupního skriptu se využívá metody `run_command`, která byla zděděna z třídy `Runner`.

6.5 Simulace testovacích scénářů

Simulace testovacích scénářů probíhá v simulačním prostředí Vivado a pomocí skripu `simulator.py`, jehož součástí je i třída `Simulator`, která režíruje samotné spouštění jednotlivých simulací. Pro spuštění simulací v prostředí Vivado se ještě využívá dodatečného skriptu `neorv32_test.tcl`, který byl převzat z [1]. Skript `neorv32_test.tcl` byl modifikován a byl rozšířen o následující argumenty, které jsou předávány při volání skriptu v rámci Python skriptu:

- `--project_dir`: cesta k Vivado projektu

- `--input_file`: soubor ve formátu `.cvs` se vstupnímy daty
- `--software_file`: aplikační obraz
- `--output_file`: výstupní soubor simulace ve formátu `.cvs`
- `--com_outpu_file`: soubor záznamu komunikace SPI a UART ve formátu `.cvs`
- `--runtime`: délka simulace
- `--pwm`: povolení PWM

6.5.1 Testbed

Samotný testbed navržený v [1] byl rozšířen o SPI monitor, který zaznamenává každý přijatý bajt. Dále byl rozšířen o zaznamenávání časové stopy startu příjmu a konce příjmu bajtu pro komunikační rozhraní UART a SPI. Tato časová stopa společně s informací o jaké komunikační rozhraní šlo a zda šlo o začátek či konce příjmu jsou zaznamenávány do souboru výstupního souboru `com_output.cvs`. Dalším rozšířením bylo implementování driveru pro možnost ovládání externích vstupů přerušení ze vstupního konfiguračního souboru, jež je blíže popsán v 6.5.3.

Nastavení parametrů procesoru zůstalo beze změny.

6.5.2 Třída `Simulator`

Třída `Simulator` je potomkem třídy `Runner` ke kontrole potřebných souborů ke spuštění simulace, ke konfiguraci tetsbedu a k spuštění samotné simulace v prostředí Vivado. Spuštění simulace je realizováno pomocí TCL skriptu `neorv32_test.tcl`. Diagram třídy `Simulator` si můžeme prohlédnout na obrázku 6.5.

Atributy

Třída obsahuje následující atributy:

`main_path`

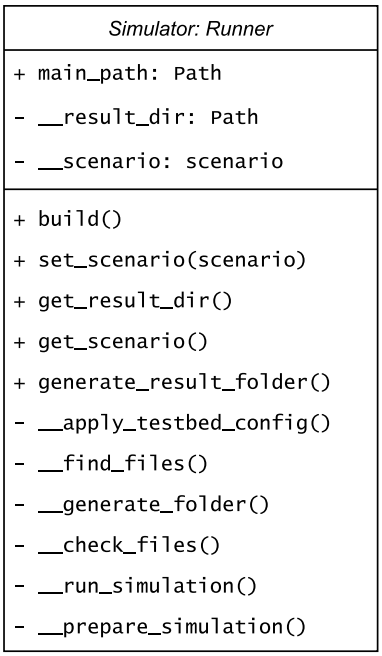
Cesta do hlavní složky.

`__result_dir`

Cesta ke složce s výstupními daty.

`__scenario`

Konfigurace aktuálně simulovaného testovacího scénáře.



Obr. 6.5: Diagram třídy `Simulator`

Metody

Součástí třídy jsou následující metody:

`build()`

Metoda `build()` je hlavní metodou objektu, která je využita v hlavním skriptu neorv32_app_simulator.py. Po zavolání této metody dojde ke spuštění simulace nastaveného testovacího scénáře.

`set_scenario(scenario)`

Tato metoda slouží k nastavení testovacího scénáře, který bude následně simulován.

`get_result_dir()`

Metoda vrací cestu ke složce s výsledky simulace.

`get_scenario()`

Metoda vrací cestu configuraci aktuálně simulovaného scénáře.

generate_result_folder()

Při zavolání této metody dojde k vygenerování výstupní složky pro testovací scénáře. Jméno složky je čas, kdy došlo k její vygenerování. Časový formát je následující **den-měsíc-rok-hodina-minuta-vteřina** a složka je vygenerována v následující lokalitě `<hlavní složka>/testbench/results`.

__apply_testbed_config()

Pomocí této metody je upraven konfigurační soubor testbedu `tb_config.sv`, který se vkládá do zdrojového souboru testbedu `neorv32_tb.sv`. Konfigurace je vyčtena z konfigurace testovacího scénáře a vyžadované nastavení je zadefinováno tak, aby bylo v souladu s pravidly jazyka System Verilog. Příklad zmíněného konfiguračního souboru si můžeme prohlédnout ve výpisu 6.4.

Výpis 6.4: Příklad konfiguračního souboru testbedu

```
parameter BAUD_RATE = 19200;
```

__find_files()

Metoda s využívá v rámci metody **__check_files()** a slouží k nalezení požadovaného souboru ve složce testovacího scénáře.

__check_files()

Metoda slouží ke zkontrolování souborů, které jsou potřebné pro korektní spuštění simulace. Těmito soubory je aplikační obraz testovacího scénáře, který musí nést název `application_image` být ve formátu `.vhd`. Tento soubor je vygenerován při překladu testovacího scénáře. Dalším vyžadovaným soubor je vstupní soubor `input_file` ve formátu `.cvs`, který bude popsán v podkapitole 6.5.3.

__generate_foler()

Ke generování výstupní složky jednotlivých testovacích scénářů se využívá následující metody. Složka vygenerována touto metodou je obsažena ve složce, která byla vygenerována metodou **generate_result_folder()** a nese název, který byl nastaven v rámci **.json** konfiguračního souboru.

__prepare_simulation()

Tato metoda slouží k připravení samotné simulace. Dochází ke kontrole souborů potřebných pro simulaci a ke generování výstupní složky testovacího scénáře.

`__run_simulation()`

Pomocí této metody je spuštěna samotná simulace testovacího scénáře za využití skriptu `neorv32_test.tcl`. Veškerý výstup běhu skriptu `neorv32_test.tcl` je zaznamenáván do souboru `simulation.log`, který je vygenerován do výstupní složky simulovaného testovacího scénáře.

6.5.3 Vstupní soubor

Vstupní soubor je ve formátu `.cvs` a jeho formát je stejný jak v práci [1]. Vstupní soubor byl rozšířen o možnost ovládání externích vstupů přerušení. Vstupní soubor příklad formátu vstupního souboru si můžeme prohlédnout ve výpisu 6.5

Výpis 6.5: Příklad vstupního souboru (uměle zarovnáno)

```
Timestamp [ ns ] ; rstn_i ; gpio_i ; irq_ch ; uart_rx
100              ; 1      ; 0      ;         ;
50100            ;        ; 1      ;         ;
80100            ;        ;        ;         ; test
2200100          ;        ; 0      ; 0001    ; off
3790100          ; 0      ;        ;         ;
```

6.5.4 Výstupní soubory

Výstupní data jsou opět ve formátu `.cvs` a stejný formát jak v práci [1]. Změnou oproti původní práci je zaznamenávání hodnoty aktuálně vykonávané instrukce. Další změnou je místo záznamu aktuální hodnoty programového čítače je zaznamenávána hodnota programového čítače, která odpovídá další instrukci v pořadí, která má být načtena, toto nám umožní ve fázi zpracovávání výsledků kontrolovat zda byla načtena očekávaná instrukce. Formát výstupního souboru si můžeme prohlédnout ve výpisu 6.6.

Výpis 6.6: Příklad výstupního souboru (první řádek)

```
Timestamp [ ns ];
fetch_pc_o [0 x ];
instr_top_o [0x];
gpio_o [0 b ];
PWM_freq [ Hz ]; PWM_duty [ x /255];
uart0_txd_o
```

Kromě výstupního souboru, který zaznamenává hodnoty výstupů, je dalším výstupním souborem soubor s názvem `com_output` taktéž ve formátu `.cvs`. Tento soubor zaznamenává časovou stopu začátku příjmu dat na komunikačních rozhraních SPI a UART a časovou stopu, kdy dojde k příjmu jednoho bajtu. Tyto záznamy o komunikaci následně slouží k výpočtu rychlosti komunikace pro jednotlivé rozhraní. Příklad tohoto výstupního souboru si můžeme prohlédnout ve výpisu 6.7.

Výpis 6.7: Příklad výstupního souboru

```
UART;[1318610 ns];start
UART;[1787530 ns];stop
```

6.6 Zpracování výstupních dat

Zpracování výstupních dat je realizováno skriptem `result_processor.py`, jehož součástí je třída `ResultProcessor`. V rámci zpracování výstupních dat se pro všechny testovací scénáře vyhodnocuje :

- rychlost komunikace SPI
- rychlost komunikace UART
- zaznamenání počtu vykonaných jednotlivých instrukcí
- zaznamenání počtu vykonaných jednotlivých funkcí
- kontrola vykonávané instrukce s očekávanou instrukcí

Dodatečně se pro FreeRTOS testovací scénáře vyhodnocuje následující:

- spočítání časové prodlevy mezi opakovaným spuštěním jednotlivých úloh

Při zpracovávání výstupních dat dochází ke generování několika výstupních souborů, názvy a formát těchto souborů je blíže popisu metod, které jsou vygenerovány do výstupní složky testovacího scénáře.

6.6.1 Třída `ResultProcessor`

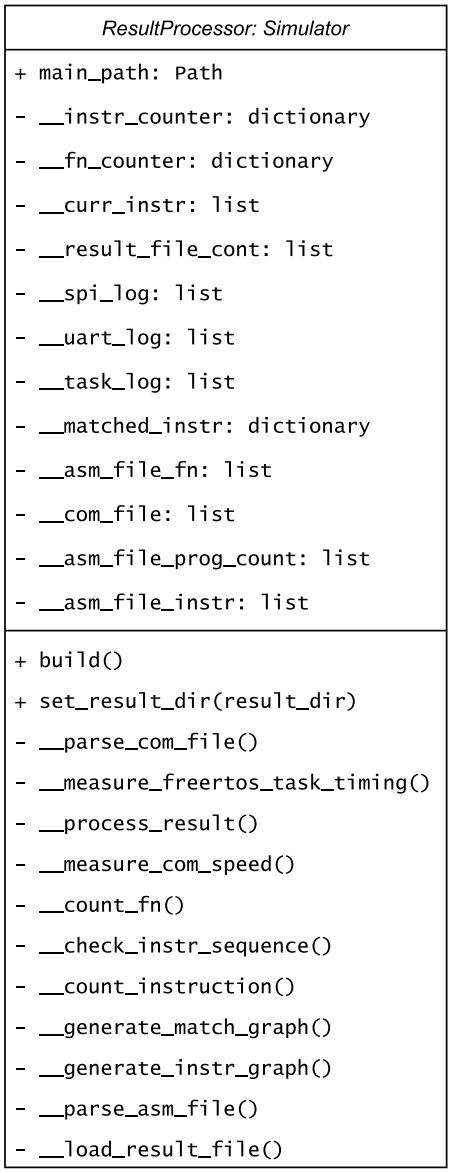
Třída `ResultProcessor` je nejobsáhlejší třídou celého frameworku a dědí atributy ze třídy `Simulator`, čily i `Runner`. Jak již bylo zmíněno, tak třída slouží ke zpracování výstupních dat ze simulace. Diagram třídy si můžeme prohlédnout na obrázku 6.6.

Atributy

Třída obsahuje následující atributy:

`__instr_counter`

Počet vykonaných jednotlivých instrukcí.



Obr. 6.6: Diagram třídy `ResultProcessor`

__fn_counter

Počet vykonaných jednotlivých funkcí.

__curr_instr

Aktuálně kontrolovaná instrukce.

__result_file_cont

Obsah výstupního souboru `output_data.cvs`.

__spi_log

Záznamy SPI komunikace.

__uart_log

Záznamy UART komunikace.

__task_log

Záznamy spouštění jednotlivých úloh ve FreeRTOS.

__matched_instr

Počet shodujících se a neshodujících se instrukcí.

__asm_file_fn

Hodnoty programového čítače, které odpovídají jednotlivým funkcím. Tyto informace jsou vyčteny z assemblerovského zdrojového souboru aplikace.

__com_file

Obsah výstupního souboru `com_output.cvs`.

__asm_file_prog_count

Hodnoty programového čítače. Index hodnoty programového čítače odpovídá instrukci v atributu `__asm_file_instr`.

__asm_file_instr

Hodnoty kódu instrukce a assemblerovského ekvivalentu. Tyto hodnoty jsou vyčteny z assemblerovského zdrojového souboru aplikace.

Metody

Třída obsahuje následující metody:

`build()`

Metoda `build()` je hlavní metodou objektu, která je využita v hlavním skriptu neorv32_app_simulator.py. Po zavolání této metody dojde ke zpracování výstupních dat simulace, který jsou načteny z výstupní složkyš.

`set_result_dir(result_dir)`

Tato metoda se využívá v hlavním skriptu `neorv32_app_simulator.py` a slouží k nastavení složky s výstupními daty testovacího scénáře.

`__parse_com_file()`

Pomocí této jsou je parsován výstupní komunikační soubor `com_output.cvs` a záznamy jednotlivých komunikací jsou roztříděny do atribut `__spi_log` a `__uart_log` v závislosti na tom zda se jedná o SPI či UART komunikační záznam.

`__process_result()`

Tato metoda slouží k postupnému spouštění metod, které zpracovávají výstupní data simulace. V rámci této metody je také pro každou vykonávanou instrukci nalezen její ekvivalent v assemblerovském zdrojovém kódu testovacího scénáře. Dochází tedy k rozšíření výstupního souboru a výsledek je zapsán do nového výstupního souboru `output_data_processed.cvs`.

`__measure_com_speed()`

Pomocí této metody dochází k výpočtu rychlosti komunikace SPI a UART. Princip výpočtu je blíže popsán v podkapitolách 6.6.2 a 6.6.3. Spočítané hodnoty pro UART jsou zapsány do souboru `uart_speed.txt` a pro SPI do souboru `spi_speed.txt`. Příklad záznamu zapsaného do výstupního UART souboru si můžeme prohlédnout ve výpisu 6.8 a pro SPI ve výpisu 6.9.

Výpis 6.8: Příklad výstupního souboru rychlosti komunikace UART (jeden řádek)

```
Transaction no. 1
Start time: 1318610ns,
Stop time: 1787530ns,
Transaction time: 468920ns,
Time to send 1 bit: 52102.2ns,
```

```
Bits per sec: 19193.0
```

Výpis 6.9: Příklad výstupního souboru rychlosti komunikace SPI (jeden řádek)

```
Transaction no. 1
Start time: 204150ns,
Stop time: 204710ns,
Transaction time: 560ns,
Time to send 1 bit: 70.0ns,
Bits per sec: 14285714.285714285
```

`__count_fn()`

Pomocí této metody je spočítán počet vykonaných jednotlivých funkcí během běhu simulace. Princip spočívá v porovnávání aktuální hodnoty programového čítače z výstupního souboru s hodnotami programového čítače z assemblerovského souboru, kdy pokud kontrolovaná hodnota programového čítače odpovídá takové hodnotě, jež odkazuje na první instrukci v některé z funkcí programu, je k celkové hodnotě vykonání této funkce přičtena jednička. Při počítání funkcí u FreeRTOS scénářů je každý záznam o spuštění FreeRTOS úlohy zaznamenán do atributy `__task_log()` společně s časovou stopou. Tohoto se využívá při výpočtu prodlevy mezi jednotlivými úlohami.

`__check_instr_sequence()`

V této metodě dochází ke kontrole předpokládané instrukce s aktuálně vykonávanou instrukcí. Neshody jsou zapisovány do výstupního souboru `not_matched.txt`. Příklad si můžeme prohlédnout ve výpisu 6.10. Princip spočívá v nalezení instrukce, která odpovídá hodnotě programového čítače a následné kontrole, zda v dalším řádku výstupního souboru se vykonává tato očekávaná instrukce.

Výpis 6.10: Příklad výstupního souboru neshod instrukcí (jeden řádek)

```
Instruction not matched!
Time: 12100 - 12130,
Program counter value: 00000180 ,
Expected instruction code : 00812423 ,
Actual Instruction : xxxxxxxx
```

__measure_freertos_task_timing()

Metoda slouží k výpočtu prodlevy mezi spuštěním jednotlivých úloh ve FreeRTOS testovacích scénářích. Výsledky je zapsány do souboru `task_timing.txt`. Příklad výsledku si můžeme prohlédnout ve výpisu 6.12. Jakým způsobem se počítá prodleva je blíže rozvedeno v podkapitole 6.6.4.

Výpis 6.11: Příklad výstupního souboru prodlev mezi FreeRTOS úlohami

```
Task: Task1     Time between task start: 11215890ns
```

__count_instruction()

Následující metoda se využívá k spočítání vykonání jednotlivých instrukcí během běhu simulace. Počet vykonaných jednotlivých instrukcí je následně zapsán do souboru `instruction_count.cvs`. Princip počítání instrukcí je obdobný jak u počítání funkcí s tím rozdílem, že zaznamenán počet každé vykonané jednotlivé instrukce instrukce.

Výpis 6.12: Příklad výstupního souboru počtu instrukcí

```
lui;125
auipc;204
addi;1083
```

__generate_match_graph()

Pomocí této metody je generován graf, který zobrazuje počet shodnutých/neshodnutých instrukcí.

__generate_instr_graph()

Metoda slouží ke generování grafu, jež zobrazuje počet jednotlivých vykonaných instrukcí.

__parse_asm_file()

Metoda se využívá k naparsování assemblerovského souboru testovacího scénáře.

__load_result_file()

Tato metoda slouží k načtení výstupních souborů simulace `output_data.cvs` a `com_output.cvs`.

6.6.2 Výpočet rychlosti komunikace SPI

Při postupu výpočtu komunikační rychlosti SPI se postupuje následovně :

1. Jelikož je znám čas začátku a ukončení přijímání bajtu, je na základě těchto hodnot spočítán čas transakce $\Delta time$ pomocí vzorce 6.1.
2. Z výsledné hodnoty $\Delta time$ je spočítána hodnota potřebná k odeslání jednoho bitu pomocí vzorce 6.2.
3. Ze získaného času potřebného pro odeslání jednoho bitu se pomocí vzorce 6.3 spočítá počet bitů odeslaných během jedné vteřiny.

$$\Delta time = stop\_time - start\_time[ns] \tag{6.1}$$

$$t\_per\_bit = \frac{\Delta time}{8}[ns] \tag{6.2}$$

$$bits\_per\_s = \frac{1}{t\_per\_bit * 10^{-9}}[bit/s] \tag{6.3}$$

6.6.3 Výpočet rychlosti komunikace UART

Při postupu výpočtu komunikační rychlosti UART je postupováno obdobně jak při výpočtu rychlosti SPI, postupuje se následovně :

1. Je znám čas začátku a ukončení přijímání bajtu, na základě těchto hodnot spočítán čas transakce $\Delta time$ pomocí vzorce 6.1.
2. Z výsledné hodnoty $\Delta time$ je spočítána hodnota potřebná k odeslání jednoho bitu pomocí vzorce 6.4, musíme však brát v potaz, že i přes to, že start komunikace je zaznamenán až po přijetí start bitu, záznam ukončení přijímutí bajtu je zaznamenán až po přijetí stop bitu, je tedy nutné stop bit zahrnout do výpočtu.
3. Ze získaného času potřebného pro odeslání jednoho bitu se pomocí vzorce 6.5 spočítá počet bitů odeslaných během jedné vteřiny.

$$t\_per\_bit = \frac{\Delta time}{9}[ns] \tag{6.4}$$

$$bits\_per\_s = \frac{1}{t\_per\_bit * 10^{-9}}[bit/s] \tag{6.5}$$

6.6.4 Výpočet časové prodlevy mezi FreeRTOS úlohami

Při výpočtu časové prodlevy mezi FreeRTOS úlohami dochází v podstatě k jednoduchému rozdílu času pomocí vzorce 6.1. Je však nutné mít na paměti, že existuje možnost, že v rámci FreeRTOS režie se do funkce vstoupí znovu a to po velmi krátké

době od jejího ukončení a je tedy nutné tyto vstupy ignorovat jelikož nedochází k vykonání funkce jako takové.

6.7 Použití

V následující podkapitole se seznámíme s principem použití samotného frameworku. Jak již bylo zmíněno, tak hlavním skriptem je `neorv32_app_simulator.py`, jehož vstupní argumenty se můžeme prohlédnout v tabulce 6.1.

Vstupní volby frameworku	Popis funkcionality
`--help (-h)`	nápověda
`--generate_project (-gp)`	generace Vivado projektu
`--project_path (-pp)`	cesta k existujícímu projektu
`--vivado_path (-vp)`	cesta k programu Vivado
`--simulate (-s)`	simulace jednoho scénáře
`--simulate_all (-sa)`	simulace všech scénářů

Tab. 6.1: Vstupní volby testovacího frameworku

Z argumentů v tabulce 6.1 jsou následující argumenty povinné: `--generate_project` nebo `--project_path`, `--vivado_path` a `--simulate` nebo `--simulate_all`. Ve výpisu 6.13 si můžeme prohlédnout výpis nápovědy hlavního skriptu.

Výpis 6.13: Výpis nápovědy

```
options:
  -h, --help            show this help message and exit
  -gp PATH, --generate_project PATH
                        Path to location where Vivado
                        project will be generated.
  -pp PATH, --project_path PATH
                        Path to Noerv32 Vivado project.
  -vp PATH, --vivado_path PATH
                        Path to Vivado install directory.
  -s PATH, --simulate PATH
                        Path to simulated scenario.
  -sa, --simulate_all   Simulate all test scenarios.
```

Veškeré aktivity je možno pozorovat v příkazovém řádku pomocí výpisů, který skript generuje během svého běhu. Výpis je také ukládán do záznamového souboru `neorv32_app_simulator.log`.

Příklad volání skriptu v operačním systému Windows si můžeme prohlédnout ve výpisu 6.14.

Výpis 6.14: Příklad volání skriptu ve Windows

```
python3 .\neorv32_app_simulator.py `
   --vivado_path D:\Xilinx\Vivado\2022.1\ `
   --generate_project .\project\ `
   --simulate .\testbench\test_scenarios\uart_loopback
```

7 Implementace testovacích scénářů

V následující kapitole si ve zkratce přiblížíme implementaci jednotlivých testovacích scénářů, které se využívají pro vyhodnocování dat.

Testovací scénáře slouží k demonstrace vlastností frameworku. FreeRTOS testovací scénáře slouží především k možnému určení, jakým způsobem by mohl být detekován nežádoucí stav z výstupních zpracovaných dat.

7.1 SPI

SPI testovací scénář `demo_spi` se skládá z odeslání 10 bajtů pomocí SPI výstupu. Na základě tohoto testovacího scénáře je spočítána komunikační rychlost SPI.

Tento testovací scénář je spouštěn několikrát, a to za účel změření několika dostupných komunikačních rychlostí, kdy je následně změřená komunikační rychlost porovnána s předpokládanou rychlostí. Děličku hodin pro SPI je možno nastavit na následující hodnoty a to : 2, 4, 8, 64, 128, 1024, 2048 a 4096.

Z demonstračního zdrojového kódu autorů procesoru víme, že komunikační rychlost SPI je spočítána dle vzorce 7.1.

$$f_{spi} = \frac{f\_cpu}{2 * CLK\_SCALER}[Hz] \tag{7.1}$$

Jelikož je SPI plně synchronním komunikačním protokolem, můžeme předpokládat, že výsledná frekvence se rovná počtu bitů odeslaných během jedné vteřiny. Předpokládané hodnoty komunikačních rychlostí při hodinovém signálu procesoru 100MHz si můžeme prohlédnout v tabulce 7.1.

Dělička hodin	**Předpokládaná rychlost [bit/s]**
2	25 000 000
4	12 500 000
8	6 250 000
64	781 250
128	390 625
1024	48 828
2048	24 414
4096	12 207

Tab. 7.1: Předpokládaná rychlost komunikace SPI

Pro tento testovací scénář budou v kapitole 8.3 diskutovány pouze hodnoty spočítané komunikační rychlosti a nebude kladen důraz na ostatní zaznamenané parametry.

7.2 UART

UART testovací scénář `uart_loopback` převzatý z [1] je využit primárně k měření rychlosti UART komunikace. Scénář byl nakonfigurován na následující hodnoty baud rate : 4800, 9600, 19200, 38400 a 115200. V rámci testovaného scénáře je přes vstupní soubor odeslaná zpráva "`uart_loopback`".

7.3 Přerušení

Pro demonstraci funkcionality driveru pro ovládání externích přerušení implementovaném v rámci testbedu slouží testovací scénář `demo_xirq`, který vytiskne přes UART hodnotu kanálu, přes který bylo vyvoláno přerušení.

7.4 FreeRTOS testovací scénáře

Pro FreeRTOS byly navrženy takové velmi jednoduché testovací scénáře, které mají za úkol odsimulovat a demonstrovat některý z nežádoucích stavů, které mohou v rámci FreeRTOS programů vzniknout. K několika testovacím scénářům, kde by se tato chyba nemusela projevit na první pohled v zpracovaných datech, byly vytvořeny i co nejpodobnější testovací scénář, který tuto chybu neobsahují, a to za účelem následného porovnání zpracovaných dat a následném určení rozdílu.

Deadlock

K simulování deadlocku slouží testovací scénář `freeRTOS_deadlock`, jeho protějškem je následovně `freeRTOS_no_deadlock`.

K simulaci deadlocku dochází následovně : v testovacím scénáři existují dvě úlohy a to `Task1` s prioritou nižší a `Task2` s prioritou vyšší a dva mutexy `xMutex` a `yMutex`. Je vytvořen a spuštěn `Task1`, který si vezme `xMutex` a vytvoří `Task2`, následně dochází ke spuštění `Task2` jelikož má vyšší prioritu. `Task2` si vezme `yMutex` a snaží se dál získat `xMutex`, který vlastní úloha `Task1`.

Program se tedy vrací zpátky do úlohy `Task1`, která se v dalším kroku snaží pro sebe získat `yMutex`, který vlastní úloha `Task2`. Ani jedna úloha tak nemůže pokračovat ve svém vykonávání jelikož obě čekají na to až mutexy vzájemně uvolní. Dochází tedy k deadlocku.

Pro scénář `freeRTOS_no_deadlock` je s mutexy zacházeno bezpečněji, kdy po vytvoření `Task2` a návratu do úlohy `Task1` je mutex `xMutex` uvolněn a program může dále pokračovat ve svém vykonávání.

Obecně vytváření úloh uvnitř jiných úloh může být nebezpečné jelikož může dojít k přetečení paměti z toho důvodu je `Task2` vždy ukončen na konci úlohy `Task1`.

Vyhladovění

Při vyhladovění dochází k situaci, kdy některé z úloh není poskytnut čas k vykonávání. K simulaci tohoto nežádoucího stavu slouží scénáře `freeRTOS_starvation` a `freeRTOS_no_starvation`.

Simulace starvation probíhá poměrně jednoduše. Jsou vytvořeny dvě úlohy, `Task1`, která má za úkol nastavovat GPIO port, s nižší prioritou a úloha `Task2` s prioritou vyšší. Jako první dojde ke spuštění úlohy `Task2` v rámci, které je nekonečná smyčka, která v tomto případě simuluje velmi dlouhé vykonávání úlohy. `Task1` se nikdy nemůže vykonat a dojde tedy k vyhladovění.

Race condition

Pro simulaci nežádoucího stavu známého jako race condition vznikly dva testovací scénáře, a to `freeRTOS_race_condition` a `freeRTOS_no_race_condition`. V obou těchto případech se vytvářejí dvě úlohy se stejnou prioritou, kdy obě úlohy přistupují ke globální proměnné, načtou si její hodnotu, inkrementují ji, následně vyčkají krátko náhodnou dobu a nakonec tuto inkrementovanou hodnotu opět uloží do globální proměnné.

Ve scénáři `freeRTOS_no_race_condition` je implementován mutex, který si úloha vezme před přístupem k této globální proměnné a po ukončení práce s ní ho uvolní. V tomto případě tedy nedochází k race condition, jelikož ke globální proměnné přistupuje pouze jedna z úloh.

Unbounded inverse priority

K demonstraci inverze priority slouží testovací scénář `freeRTOS_bprio_inversion`. K inverzi dochází podobným způsobem jako k vyhladovění. V rámci scénáře jsou vytvořeny dvě úlohy, a to `Task1` s prioritou nižší a `Task2` s prioritou vyšší. Dále vytvořen mutex `xMutex`. Po spuštění úloh se první začne vykonávat úloha `Task2`, která si vezme mutex, nastaví GPIO port a mutex vrátí, následně se začne vykonávat úloha `Task1`, která si opět vezme mutex a následně se zacyklí v nekonečné smyčce, čímž simulujeme například zacyklení samotného programu uvnitř úlohy, případně extrémně dlouhé vykonávání. I přes to, že `Task2` má vyšší prioritu, tak se nemůže

vykonávat, jelikož čeká na mutex od úlohy `Task1` a dochází tedy k inverzi priority, kdy úloha s prioritou vyšší je závislá na úloze s prioritou nižší.

8 Výsledky testovacích scénářů

V následující kapitole budou diskutovány výsledky jednotlivých testovacích scénářů a vlastnosti samotné simulace. Vlastnostmi simulace myslíme simulační čas, reálnou délku simulace a maximální využití RAM paměti. Tyto vlastnosti jsou získány přímo z Vivado simulátoru.

Každý testovací scénář generuje velké množství výstupních dat, a proto jsou pro každý scénář jsou vytipovány určité výstupní data, ve kterých je dobře viditelná či čitelná testovaná situace.

8.1 Rychlost komunikace SPI

Jak již bylo zmíněno, tak pro měření rychlosti SPI komunikace byl využít primárně SPI testovací scénář `demo_spi`. Bylo testováno několik komunikačních rychlostí, kdy spočítané hodnoty společně s odchylkou od teoretické hodnoty si můžeme prohlédnout v tabulce 8.1, kde S_p je předpokládá komunikační rychlost a S_s je spočtená komunikační rychlost. Hodnoty spočítaného komunikační rychlosti byly převzaty z výstupního souboru testovacího scénáře `spi_speed.txt`.

Dělička hodin	S_p **[bit/s]**	S_s **[bit/s]**	**Rozdíl [bit/s]**
2	25 000 000	28 571 428	3 571 428
4	12 500 000	14 285 714	1 785 174
8	6 250 000	7 142 857	892 857
64	781 250	892 857	111 607
128	390 625	446 428	55803
1024	48 828	55 803	6975
2048	24 414	27 901	3487
4096	12 207	13 950	1743

Tab. 8.1: Tabulka porovnání SPI komunikační rychlosti

Z tabulky je zřejmé, že komunikační rychlost se v žádném z případů neblíží předpokládané komunikační rychlosti. Na vině by mohla být dělička hodinového signálu, jejíž výstup slouží jako zdroj hodinového signálu SPI a bylo by vhodné tuto komponentu blíže prozkoumat a pokusit se určit, zda je zdrojem této chyby.

V tabulce 8.2 si můžeme prohlédnout vlastnosti simulace pro testovací scénář `demo_spi`.

Scénář	Dělička hodin	Simulovaný čas	Simulační čas	Max. RAM
`demo_spi`	2	10ms	48s	1295 MB
`demo_spi`	4	10ms	48s	1295 MB
`demo_spi`	8	10ms	47s	1294 MB
`demo_spi`	64	10ms	49s	1294 MB
`demo_spi`	128	10ms	49s	1293 MB
`demo_spi`	1024	10ms	56s	1294 MB
`demo_spi`	2048	10ms	65s	1294 MB
`demo_spi`	4096	10ms	82s	1294 MB

Tab. 8.2: Vlastnosti simulace `demo_spi`

8.2 Rychlost komunikace UART

V rámci testovacího scénáře `uart_loopback` byl baud rate nastaven na následující hodnoty : 4800, 9600, 19200, 38400 a 11520.

V tabulce 8.2 si můžeme prohlédnout naměřené rychlosti pro UART komunikaci v rámci testovacích scénářů. Hodnoty spočítaného baud rate byly převzaty z výstupního souboru testovacího scénáře `uart_speed.txt`.

Předpokládaný baud rate	Spočítaný baud rate	Rozdíl [bit/s]
4800	4799	1
9600	9598	2
19 200	19 193	7
38 400	38 369	31
115 200	114 913	7

Tab. 8.3: Tabulka porovnání UART komunikační rychlosti

Z tabulky je zřejmé, že rozdíly v komunikační rychlosti jsou minimální až zanedbatelné a můžeme prohlásit, že rychlost komunikace odpovídá předpokládaným hodnotám a UART je tedy přesný.

V tabulce 8.2 si můžeme prohlédnout vlastnosti simulace pro testovací scénář `uart_loopback`.

8.3 Přerušení

Při simulaci testovacího scénáře `demo_xirq` sloužícího k demonstraci funkcionality driveru externího přerušení jsme mohli zjistit, že driver funguje správně a při vyvolání přerušení na určitém vstupu je na výstup UART vytisknut hodnota kanálu,

Scénář	Baud rate	Simulovaný čas	Simulační čas	Max. RAM
uart_loopback	4800	10ms	99s	1293 MB
uart_loopback	9600	10ms	100s	1294 MB
uart_loopback	19200	10ms	99s	1294 MB
uart_loopback	38400	10ms	99s	1294 MB
uart_loopback	115200	10ms	98s	1294 MB

Tab. 8.4: Vlastnosti simulace uart_loopback

která odpovídá hodnotě kanálu, na kterém bylo přerušení vyvoláno. Tuto skutečnost si můžeme ověřit ve výpisu 8.1 z výpisového souboru simulace simulation.log

Výpis 8.1: Výpis z simulation.log testovacího scénáře demo_xirq

```
[100 ns]  rstn_i    = [0b1]
[41350 ns]  gpio_o    = [0b00000000]
[200000 ns] Triggering xirq = 0001
[400000 ns] Triggering xirq = 0010
[600000 ns] Triggering xirq = 0100
[676430 ns] uart_tx_o = ['0']
[800000 ns] Triggering xirq = 1000
[1197270 ns]  uart_tx_o = ['1']
[1718110 ns]  uart_tx_o = ['2']
[2238950 ns]  uart_tx_o = ['3']
```

V tabulce 8.3 si můžeme prohlédnout vlastnosti simulace pro testovací scénář demo_xirq.

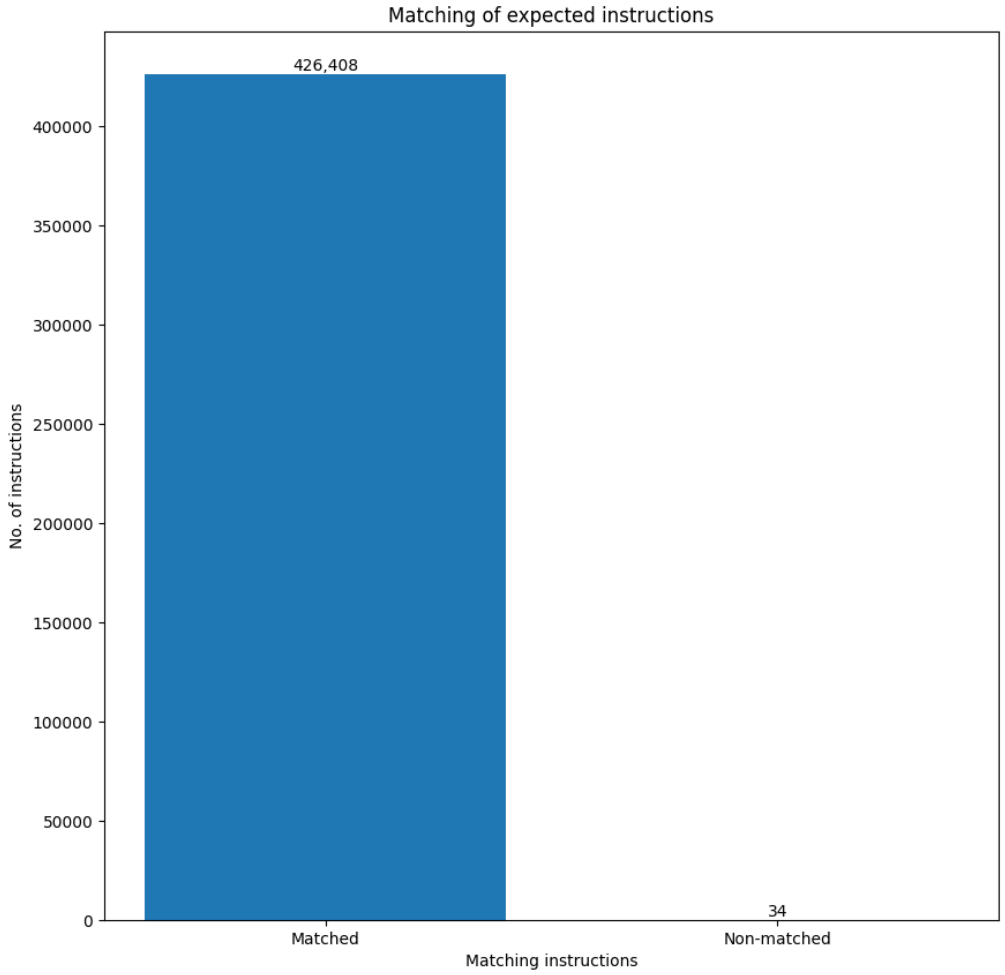
Scénář	Simulovaný čas	Simulační čas	Max. RAM
demo_xirq	10ms	93s	1286 MB

Tab. 8.5: Vlastnosti simulace demo_xirq

8.4 Kontrola správné sekvence instrukcí

K demonstraci grafu shod instrukcí můžeme využít testovací scénář demo_xirq. Na obrázku si 8.4 si tento graf můžeme prohlédnout.

Z grafu můžeme vyčíst, že došlo celkově ke 34 neshodám s očekávanou instrukcí, kdy bližší informace získáme ze souboru not_matched.txt. Výpis zobrazující jeden řádek tohoto souboru si můžeme prohlédnout ve výpisu 8.2



Obr. 8.1: Graf vykonaných instrukcí scénáře `demo_xirq`

Výpis 8.2: Počet spouštění úloh pro scénář simulující deadlock

```
Instruction not matched!
Time: 12100 - 12130,
Program counter value: 00000180 ,
Expected instruction : 00812423 ,
Actual Instruction : xxxxxxxx
```

Ve všech případech, kdy byla zaznamenána neshoda, je na vině, že hodnota instrukce je nepředvídatelná, což se projevuje v simulaci právě zaznamenanými hodnotami `xxxxxxxx`. Tato chyba může být způsobena samotným RTL zdrojovým kódem, kdy v některých okamžicích může být do signálu aktuálně vykonávající se instrukce přiřazováno ve více úsecích kódu zároveň, což způsobuje tuto nepředvídatelnou hodnotu. Pro přesné detekování problému by byla potřeba detailnější analýza RTL zdrojového kódu.

8.5 FreeRTOS scénáře

FreeRTOS scénáře slouží k demonstraci schopností testovacího frameworku a k možnému určení, jak by některý z nežádoucích stavů mohl být detekován při simulaci složitějších scénářů.

8.5.1 Deadlock

Jelikož při deadlocku dochází k zasekutní celého programu můžeme očekávat, že celkové spuštění jednotlivých úloh bude nižší, než je předpokládáno. Jelikož simulace byla spuštěno po dobu 50ms, kdy prodleva mezi úlohami je 10ms, očekáváme tedy, že dojde alespoň k pěti spuštěním jednotlivých úloh.

Výpis počtu spouštění úloh pro deadlock scénář z výsledného souboru `function_count.cvs` si můžeme prohlédnout ve výpisu 8.3.

Výpis 8.3: Počet spouštění úloh pro scénář simulující deadlock

```
<Task1>:;1
<Task2>:;1
```

Výpis počtu spouštění úloh pro scénář neobsahující deadlock z výsledného souboru `function_count.cvs` si můžeme prohlédnout ve výpisu 8.4.

Výpis 8.4: Počet spouštění úloh pro scénář bez deadlocku

```
<Task1>:;6
<Task2>:;5
```

Jak bylo předpokládáno, tak počet spuštění úloh pro deadlock scénář je razantně nižší. Deadlock by tedy pro komplexnější scénáře mohl být detekován pomocí neobvykle nízkého počtu spuštění jednotlivých funkcí.

V tabulce 8.5.1 si můžeme prohlédnout vlastnosti simulace simulovaných scénářů.

Sscénář	Simulovaný čas	Simulační čas	Max. RAM
`freeRTOS_deadlock`	50ms	234s	1294 MB
`freeRTOS_no_deadlock`	50ms	241s	1295 MB

Tab. 8.6: Vlastnosti simulace `freeRTOS_deadlock` a `freeRTOS_no_deadlock`

8.5.2 Race condition

Navržený testovací scénář, který demonstruje race condition využívá převážně přístup ke globální proměnné a následnou manipulací s ní. Lze tedy očekávat, že v případě, kdy není použit mutex, by mohly být instrukce, které slouží k načítání hodnot z paměti či ukládání hodnot do paměti více využívány než v případě, kdy přístup ke globální proměnné je blokován mutexem.

Jelikož úlohy, které se využívají v těchto testovacích scénářích neobsahují prodlevu a není tak třeba delšího času simulace, byl čas simulace nastaven na 10ms.

Z vygenerovaných grafů 8.5.2 a 8.5.2, které zobrazují počet jednotlivých vykonaných instrukcí během simulace testovacího scénáře můžeme zjistit, že pro scénář, kde dochází k race condition je zvýšený počet instrukcí, které pracují s pamětí a to konkrétně instrukce `sw` (store word) a `lw` (load word).

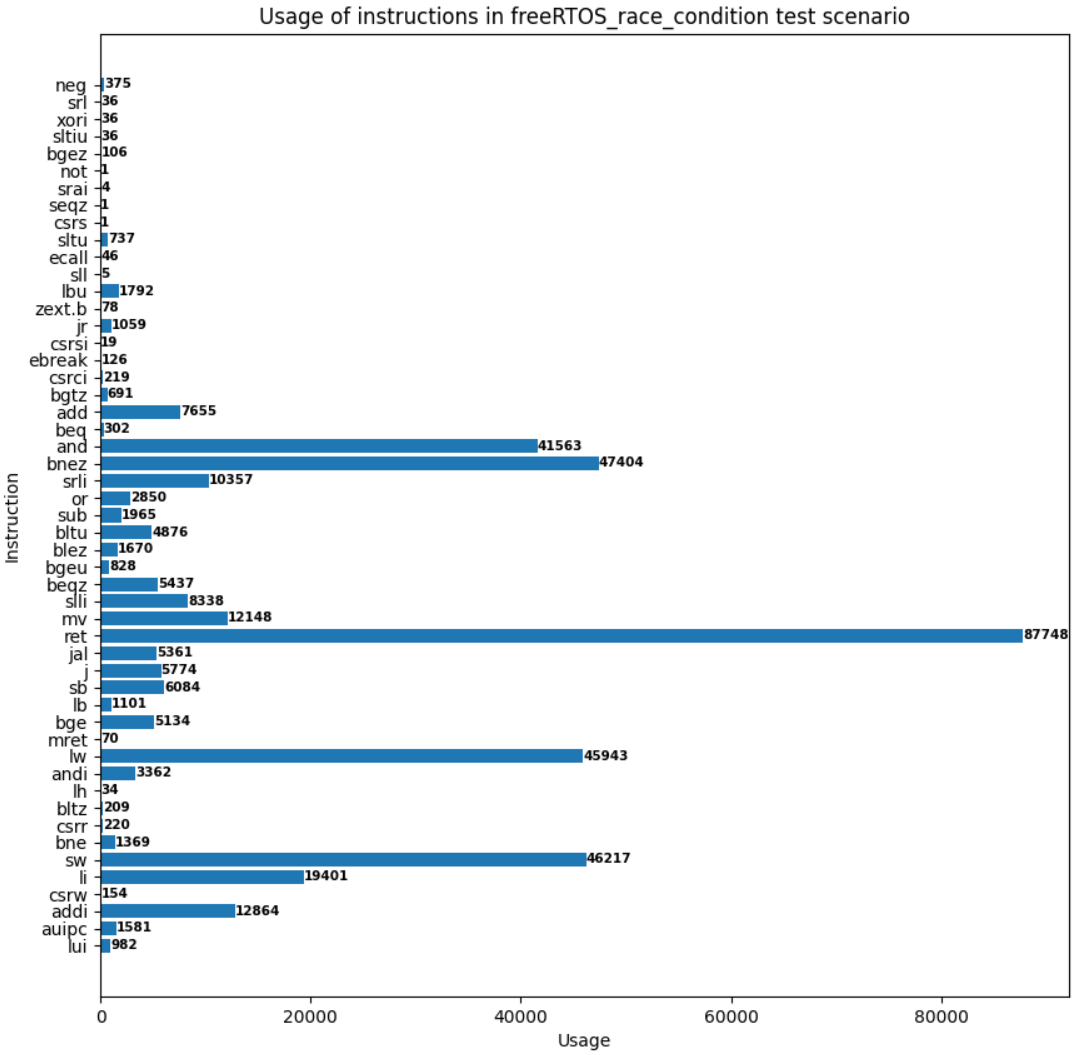
V tabulce 8.5.2 si můžeme prohlédnout vlastnosti simulace simulovaných scénářů.

Scénář	Simulovaný čas	Simulační čas	Max. RAM
`freeRTOS_race_condition`	10ms	99s	1294 MB
`freeRTOS_no_race_condition`	10ms	99s	1294 MB

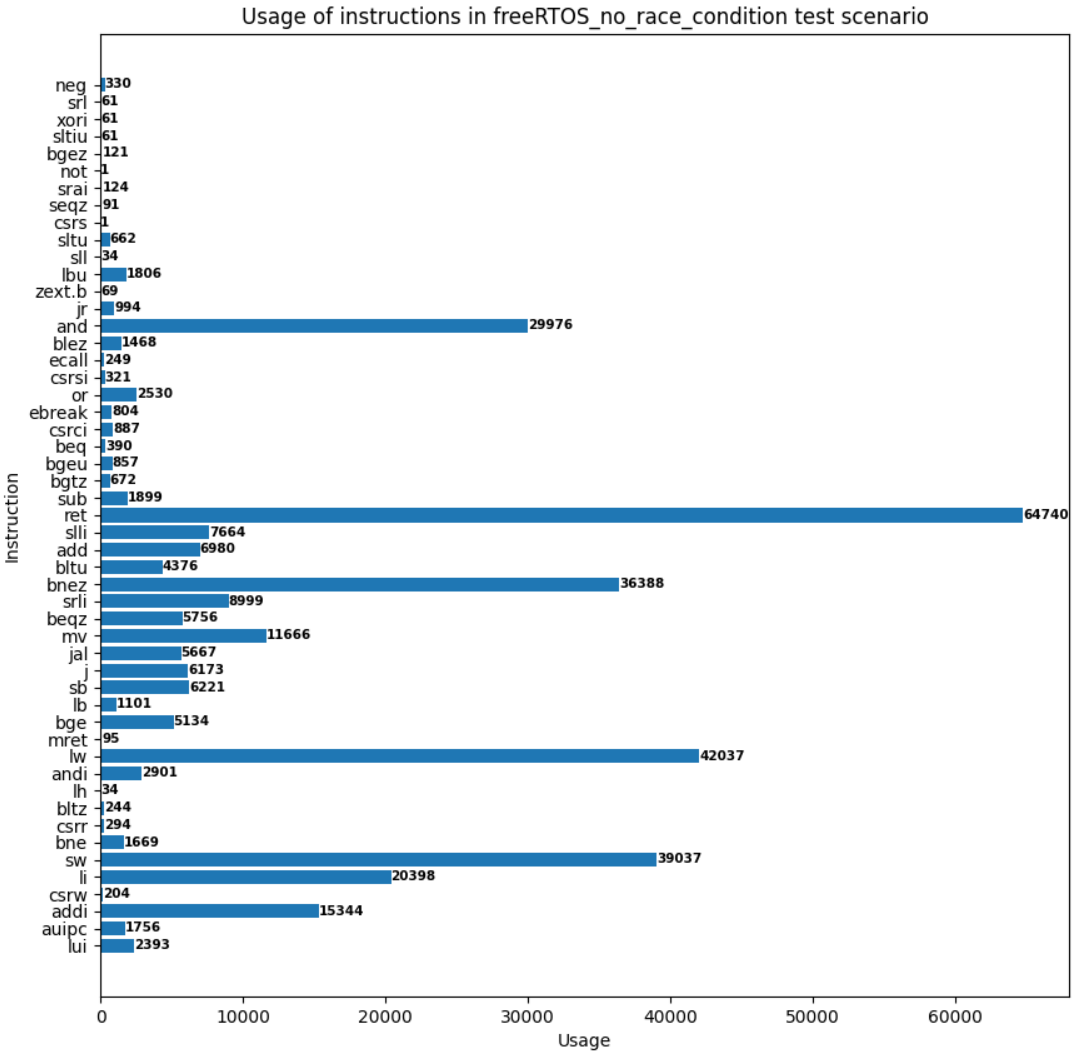
Tab. 8.7: Vlastnosti simulace pro `race_condition` scénáře

8.5.3 Vyhladovění

Při vyhladovění můžeme opět zkontrolovat výstupní soubor s počtem vykonaných jednotlivých funkcí a zkontrolovat počet kolikrát byla vykonány úloha `Task1` a



Obr. 8.2: Graf vykonaných instrukcí scénáře `freeRTOS_race_condition`



Obr. 8.3: Graf vykonaných instrukcí scénáře `freeRTOS_no_race_condition`

Task2. Jelikož prodleva obou úloh byla nastavena na 10ms a běh simulace byl nastaven na 50ms můžeme předpokládat, že úlohy budou spuštěny alespoň pětkrát.

Pri bližší analýze, zda k vyhladovění skutečně došlo se můžeme přesvědčit kontrolou výstupního souboru output_data.cvs a přesvědčit se o tom, že k vyhladovění skutečně došlo, jelikož hodnota GPIO po celou dobu simulace je 00000000.

Výpis počtu spouštění úloh pro scénář simulující vyhladovění z výsledného souboru function_count.cvs si můžeme prohlédnout ve výpisu 8.5.

Výpis 8.5: Počet spouštění úloh pro scénář bez deadlocku

```
<Task2>:;699762
<Task1>:;699761
```

Z výpisu 8.5 můžeme vidět, že došlo k neobvykle velkému počtu vstupů do úloh Task1 a Task2. Tato skutečnost může být způsobena samotnou režií FreeRTOS, který se snaží vstoupit do úlohy Task1 a vykonat ji, ale tato snaha je neustále přerušována úlohou Task2, která ještě nebyla ukončena a chce pokračovat v běhu. Úloze je v tomto případě vyhověno, jelikož má vyšší prioritu.

V tabulce 8.5.3 si můžeme prohlédnout vlastnosti simulace simulovaného scénáře.

Scénář	Simulovaný čas	Simulační čas	Max. RAM
freeRTOS_starvation	50ms	464s	1294 MB

Tab. 8.8: Vlastnosti simulace freeRTOS_starvation

8.5.4 Unbounded priority inverze

Při inverzi priority můžeme očekávat, že během běhu bude program snažit spustit úlohu s vyšší prioritou, opět tedy můžeme využít výstupní soubor function_count.cvs testovacího scénáře freeRTOS_uprio_inversion, kdy počet spuštění funkce Task2 si můžeme prohlédnou ve výpisu 8.6.

Výpis 8.6: Počet spuštění úlohy Task2 ve freeROTS_uprio_inversion

```
<Task2>:;541160
```

Jak jsme předpokládali, tak dochází k velkému počtu vstupů do tohoto úkolu, ale úloha není nikdy vykonána, jelikož je blokována úlohou s nižší prioritou. Dochází tedy ve své podstatě i k vyhladovění. Tato chyba ovšem může mít kritické následky, protože úkol s vyšší prioritou může mít za úkol vykonávat činnost na které je závislý správný chod celé aplikace.

V tabulce 8.5.4 si můžeme prohlédnout vlastnosti simulace simulovaného scénáře.

Scénář	Simulovaný čas	Simulační čas	Max. RAM
`freeRTOS_uprio_inversion`	50ms	529s	1285 MB

Tab. 8.9: Vlastnosti simulace `freeRTOS_uprio_inversion`

8.5.5 Prodleva mezi úlohami

K získání prodlevy mezi úlohami můžeme využít testovací scénář `demoFreeRTOS_no_deadlock`, který neobsahuje žádný nežádoucí stav a můžeme považovat, že má korektní chování. K zjištění prodlevy můžeme využít výstupní soubor `task_timing.txt`, jehož obsah si můžeme prohlédnout ve výpisu 8.7.

Nastavení prodlevy mezi úlohami bylo 10ms a simulace byla opět spuštěna po dobu 50ms. Můžeme tedy očekávat, že úloha bude spuštěna přibližně po 10ms což odpovídá 10 000 000ns. Z výpisu je zřejmě, že až na velmi malé odchylky je toto časování spouštění úloh velmi přesné.

Výpis 8.7: Prodleva mezi spuštěním jednotlivých úloh

```
Task: <Task1>:  time between task start: 10149900ns
Task: <Task1>:  time between task start: 10000000ns
Task: <Task1>:  time between task start: 10000000ns
Task: <Task1>:  time between task start: 10000000ns
Task: <Task2>:  time between task start: 9977420ns
Task: <Task2>:  time between task start: 10000000ns
Task: <Task2>:  time between task start: 10000000ns
```

9 Závěr

V rámci diplomové práce jsme krátce seznámení s architekturou procesorů RISC-V, principem funkcionality operačních systémů reálného času a s testovaným procesorem NOERV32.

Jelikož práce navazuje na práci [1], jsou rozebírány i výsledky této práce, na jejímž základu je následně navržen a implementován testovací framework napsán `neorv32_app_simulator` v jazyce python, který umožňuje jednoduše konfigurovat a spouštět aplikace (testovací scénáře) v soft-core RISC-V procesoru v simulačním prostředí Vivado. Kromě jednoduchého použití, jednoduché možnosti konfigurovat testovací scénáře, možnosti odebírání a přidávání testovacích scénářů do receptu simulovaných scénářů dochází i ke zpracování výstupních dat simulace.

Jsou zaznamenávány hodnoty na výstupu procesoru do výstupního souboru, kdy kromě hodnot výstupů periferií jsou zaznamenávány i hodnoty programového čítače, který odkazuje na instrukce, která má být načtena, tak i kód aktuálně vykonávané instrukce. V rámci zpracování dat je pro kód instrukce nalezen její assemblerovský ekvivalent, je měřena rychlost komunikace SPI a UART, je generován graf celkového počtu vykonaných jednotlivých instrukcí během běhu simulace, dále je zaznamenán počet vykonaných jednotlivých funkcí a je také prováděna kontrola, zda se vykonává očekávaná instrukce.

Dále bylo navrženo několik testovacích scénářů a to testovací scénář pro SPI, který slouží k určení rychlosti komunikace SPI a určení odchylky od očekávané hodnoty, dále je využit testovací scénář pro UART z práce [1], který slouží obdobnému účelu, a to k měření a určení odchylky rychlosti komunikace UART. V poslední řadě je navrženo několik testovacích scénářů FreeRTOS, která záměrně obsahují některou z kritických chyb, kterým může dojít ve FreeRTOS aplikacích. Těmito scénáři je scénář simulující deadlock, vyhladovění, race condition a unbounded priority inverzi. Pro FreeRTOS testovací scénáře je také měřena hodnota prodlevy mezi jednotlivými úlohami.

Pro testovací scénář SPI, bylo zjištěno, že dochází k velké odchylce od očekávané rychlosti komunikace. Na vině může být RTL implementace děličky hodinového signálu, jež slouží jako zdroj pro hodinový signál SPI. Rychlost komunikace UART je naopak přesná a dochází jen k velmi malým odchylkám a očekávané hodnoty odpovídají těm, které jsou změřeny simulací.

Při kontrole, zda se vykonávají očekávané instrukce bylo zjištěno, že v některých případech má hodnota signálu, jež obsahuje hodnotu aktuálně vykonávané instrukce nepředvídatelnou hodnotu, což může být způsobeno samotnou RTL implementací, kdy za určité situace může do signálu přiřazovány z vícero signálu naráz, což způsobí nepředvídatelnou hodnotu.

Pro FreeRTOS scénáře můžeme při pozorování deadlocku pozorovat velmi nízký počet spuštění jednotlivých úloh. Pro testovací scénář, který simuluje race condition oproti scénáři, který tuto chybu neobsahuje, můžeme pozorovat vyšší počet instrukcí, jež přistupují k paměti a ukládají hodnoty do paměti.

Skoro stejné chování můžeme pozorovat při vyhladovění a unbounded priority inverzi, a to neobvykle velký počet vstupů do úloh.

Při ověřování doby prodlevy mezi úlohami v jednom z FreeRTOS scénářů bylo pozorováno, že spouštění úlohy je až na velmi malé odchylky velmi přesné.

Literatura

[1] PRUSÁK, Lukáš. *Automatizovaný testbed pro SIL/PIL testování firmware pomocí FPGA* [online]. Brno, 2022 [cit. 2023-01-08]. Dostupné z: http://hdl.handle.net/11012/204799. Diplomová práce. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. Ústav automatizace a měřicí techniky. Vedoucí práce Jakub Arm.

[2] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.

[3] RISC-V International – RISC-V: The Free and Open RISC Instruction Set Architecture. *The RISC-V Instruction Set Manual Volume I: User-Level ISA* [online]. Květen, 2017 [cit. 10.04.2023]. Dostupné z: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[4] RISC-V International – RISC-V: The Free and Open RISC Instruction Set Architecture. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture* [online]. Prosinec, 2021 [cit. 10.04.2023]. Dostupné z: https://riscv.org/technical/specifications/

[5] RISC-V International – RISC-V: The Free and Open RISC Instruction Set Architecture. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA* [online]. Prosinec, 2019 [cit. 10.04.2023]. Dostupné z: https://riscv.org/technical/specifications/

[6] THORNTON, Scott. *RISC-V: an Open Instruction Set Architecture* [online]. Únor, 2018 [cit. 10.04.2023]. Dostupné z: https: //www.microcontrollertips.com/risc-v-open-instruction-set-architecture/

[7] NOLTING, Ing. Stephan. *The NEORV32 RISC-V Processor: Datasheet* [online]. In: . [cit. 2023-2-16]. Dostupné z: https://stnolting.github.io/neorv32/

[8] What Is A Real-Time Operating Systems (RTOS) | Wind River. Wind River Software | Safe, Secure, Reliable [online]. Copyright © 2023 Wind River Systems, Inc. [cit. 25.04.2023]. Dostupné z: https://www.windriver.com/solutions/learning/rtos

[9] What Is RTOS, How It Works, and 9 RTOS Platforms to Know | Sternum. IoT Security, Observability and Operations Platform | Sternum [online]. Copyright © Sternum 2023 [cit. 25.04.2023]. Dostupné z: https://sternumiot.com/iot-blog/crash-course-introduction-to-real-time-operating-system-rtos/

[10] Real Time Operating System (RTOS) - GeeksforGeeks. GeeksforGeeks | A computer science portal for geeks [online]. Dostupné z: https://www.geeksforgeeks.org/real-time-operating-system-rtos/

[11] What Is RTOS, How It Works, and 9 RTOS Platforms to Know | Sternum. IoT Security, Observability and Operations Platform | Sternum [online]. Copyright © Sternum 2023 [cit. 25.04.2023]. Dostupné z: https://sternumiot.com/iot-blog/crash-course-introduction-to-real-time-operating-system-rtos/

[12] Difference between Hard real time and Soft real time system - GeeksforGeeks. GeeksforGeeks | A computer science portal for geeks [online]. Dostupné z: https://www.geeksforgeeks.org/difference-between-hard-real-time-and-soft-real-time-system/

[13] Preemption (computing) - Wikipedia. [online]. Dostupné z: https://en.wikipedia.org/wiki/Preemption_(computing)

[14] Cooperative multitasking - Wikipedia. [online]. Dostupné z: https://en.wikipedia.org/wiki/Cooperative_multitasking

[15] Preemption (computing) - Wikipedia. [online]. Dostupné z: https://en.wikipedia.org/wiki/Preemption_(computing)

[16] FrontPage - Python Wiki. Python Software Foundation Wiki Server [online]. Naposledy změněno 16. 9. 2018 [cit. 2023-01-18]. Dostupné z: https://www.chipverify.com/systemverilog/systemverilog-tutorial

[17] SystemVerilog Tutorial. ChipVerify [online]. [cit. 2023-01-19]. Dostupné z: https://www.chipverify.com/systemverilog/systemverilog-tutorial

[18] Xilinx Vivado - Wikipedia. [online]. Naposledy změněno 24. 12. 2022 [cit. 2023-01-19]. Dostupné z: https://en.wikipedia.org/wiki/Xilinx_Vivado

Seznam příloh

A Obsah přiloženého média

```
./ ..... kořenový adresář přiloženého média
├── xohnut00_dp.pdf ..... diplomová práce v pdf
├── sim_results ..... adresář s výsledky simulací
│   └── ..
└── neorv32_app_simulator
    ├── FreeRTOS ..... adresář s FreeRTOS
    │   └── ..
    ├── neorv32 ..... adresář se zdrojovými soubory procesoru NEORV32
    │   └── ..
    ├── riscv_toolchain ..... toolchain k překladu testovacích scénářů
    │   └── ..
    ├── scripts
    │   ├── simulator ..... adresář s python zdrojovými soubory
    │   │   └── ..
    │   └── vivado ..... adresář s TCL zdrojovými soubory
    │       └── ..
    ├── testbench
    │   ├── results ..... adresáš pro generování výsledků simulací
    │   │   └── ..
    │   ├── test_scenarios .. adresář se zdrojovými kódy testovacích scénářů
    │   │   └── ..
    │   ├── noerv32_tb.sv ..... zdrojový soubor testbench
    │   ├── config.json ..... konfigurační soubor testovacích scénářů
    │   └── tb_config.json ..... konfigurační soubor pro testbench
    ├── .python-version ..... soubor s potřebnou python verzí
    ├── neorv32_app_simulator.py ..... testovací framework
    ├── README.md ..... anglický návod pro spuštění
    └── requirements.txt ..... seznam potřebných python knihoven
```