VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PROSTŘEDÍ PRO SPOUŠTĚNÍ TESTŮ KOMPATIBILITY RISC-V

FRAMEWORK FOR RISC-V COMPLIANCE TESTS EXECUTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MILAN SKÁLA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20947/2017/xskala09

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Skála Milan, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Prostředí pro spouštění testů kompatibility RISC-V**
Framework for RISC-V Compliance Tests Execution
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se se standardy popisujícími instrukční sadu a architekturu RISC-V a s aktuálním stavem vývoje v organizaci RISC-V Foundation.
2. Seznamte se s pojmem testy kompatibility a existujícími prostředky pro spouštění testů.
3. Navrhněte prostředí pro spouštění testů kompatibility různých typů implementací RISC-V, jako jsou např. instrukční simulátory anebo hardwarové modely.
4. Implementujte navržené prostředí s možností zobrazení výsledků kompatibility.
5. Experimentálně ověřte funkčnost prostředí pro jeden vybraný instrukční simulátor a jednu vybranou hardwarovou implementaci RISC-V procesoru.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- RISC-V Foundation, 2016. User-level RISC-V ISA specification, https://riscv.org/specifications/.
- RISC-V Foundation, 2017. Draft Privileged RISC-V ISA specification, https://riscv.org/specifications/.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zachariášová Marcela, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato práce se zabývá vytvořením návrhu a implementací frameworku pro spouštění testů kompatibility různých typů implementací architektury RISC-V. Popisuje historický vývoj této architektury, instrukční sadu a režimy procesoru, které tato architektura podporuje. Dále jsou rozebrány současné metodiky a frameworky pro testování implementované v jazyce Python. Důraz je kladen na rozbor testů kompatibility. V praktické části je proveden návrh a implementace frameworku pro spouštění testů kompatibility, jehož vstupem mohou být různé typy implementací RISC-V. Sekundárním cílem práce je vytvořit grafické uživatelské rozhraní umožňující rychlou a snadnou konfiguraci testů. Na závěr jsou zhodnoceny výsledky a diskutovány možnosti dalšího rozšíření.

Abstract

This thesis focuses on design and implementation of a testing framework for different implementation types of RISC-V architecture. It describes history, instruction set and processor modes which are supported by this architecture. Further, the current methodologies and testing frameworks implemented in Python are discussed. Emphasis is placed on the analysis of compliance tests. In the practical part, the design and implementation of a framework for execution of compliance tests for models, which can be implemented in various ways, either as an ISA simulator or a hardware model, is done. The secondary aim of the thesis is to create a graphical user interface for quick and easy test configuration. Finally, the results are evaluated and the possibilities of further development are discussed.

Klíčová slova

RISC-V, RISC, ARM, Internet věcí, IoT, procesor, CPU, testování, testování kompatibility, automatizace, Python

Keywords

RISC-V, RISC, ARM, Internet of Things, IoT, processor, CPU, testing, compliance testing, automation, Python

Citace

SKÁLA, Milan. *Prostředí pro spouštění testů kompatibility RISC-V*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marcela Zachariášová, Ph.D.

Prostředí pro spouštění testů kompatibility RISC-V

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením paní Ing. Marcely Zachariášové, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Milan Skála
23. května 2018

Poděkování

Chtěl bych poděkovat své vedoucí diplomové práce Ing. Marcele Zachariášové, Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce a společnosti Codasip za poskytnutí modelů RISC–V pro experimentování.

Obsah

Kapitola 1

Úvod

Procesor je základní komponentou každé hardwarové jednotky, která je řízena programem. Své počátky nachází ve 40. letech minulého století (ENIAC). Od té doby došlo ke značné evoluci, jak z hlediska výpočetního výkonu, velikosti, spotřeby, tak z pohledu architektury. V současnosti na trhu existuje několik architektur procesorů, z nichž každá je navržena tak, aby co nejvíce splňovala požadavky potřebné v prostředí, kde bude procesor využíván. V oblasti osobních počítačů a serverů jsou využívané procesory od amerických společností Intel a AMD. Největší podíl na trhu s přenosnými zařízeními (mobilních telefonů, tabletů, aj.) má původně britská společnost ARM. Architektury od těchto společností jsou proprietární a uzavřené, což výrazným způsobem snižuje jejich flexibilitu. Na druhé straně existují pokusy o vytvoření otevřených architektur. Příklady těchto architektur jsou OpenRISC, Zet, Amber a jiné. Tyto architektury dovolují jejich modifikaci, implementaci a následnou distribuci komukoliv. Nevýhodou ovšem je jejich zaměření na specifickou platformu, např. mobilní telefony nebo FPGA. To otevírá možnost vytvoření standardizované instrukční sady, která bude otevřená a platformně nezávislá. Jako první se touto myšlenkou začali zabývat profesoři Krste Asanović a David Patterson z Kalifornské univerzity v Berkeley, kteří chtěli takovou architekturu vytvořit. V roce 2010 ve spolupráci s dalšími zájemci zahájili projekt, který měl za cíl vznik takové platformy zejména pro akademické účely, ale také pro komerční užití. Výslednou architekturu instrukční sady pojmenovali RISC-V. Řecká číslovka "V" značí 5. generaci návrhů z Berkeley. Předchozími verzemi byly RISC-I, RISC-II, SOAR a SPUR. Dalšími významy jsou také variabilita a vektor (vektorové rozšíření instrukční sady).

Velmi důležitou fází při vývoji hardwaru je testování, které zajišťuje konzistenci mezi implementovaným modelem a specifikací. Účelem testování je mimo ověření souladu modelu se specifikací také zajistit to, aby model správně reagoval na nestandardní stavy, mezi které může patřit například nesprávný vstup od uživatele (nejčastěji programátora), chyba v komunikaci s periferními zařízeními, či vnitřními komponentami, vysoká zátěž, zotavení po havárii nebo jak funguje na různých konfiguracích. Testování lze rozdělit na funkční a nefunkční. Cílem funkčního testování je ověření, že produkt se chová tak, jak byl navržen, čili se soustředí na správné provedení implementovaných operací – zajišťuje tedy výše uvedenou konzistenci mezi modelem a specifikací (nejčastěji od zákazníka).

Na druhou stranu nefunkční testování slouží pro zjištění, zda produkt splňuje očekávání uživatele či zákazníka. Určuje například jak rychlá je odezva produktu na požadavek nebo jak dlouho trvá určitá akce. Konkrétními příklady nefunkčního testování jsou zátěžové testy, testování kompatibility, testování zabezpečení, testy použitelnosti. Většinou se tento druh testování provádí až po funkčním testování.

Cílem této práce je navrhnout a implementovat systém pro spouštění testů kompatibility architektury RISC-V. Výsledný systém by měl být použitelný pro různé implementace této architektury, mezi které může patřit instrukční simulátor nebo hardwarový model.

V první části práce je popsána architektura procesoru RISC-V včetně stručné historie vývoje architektury a popis instrukční sady a standardizovaných rozšíření. Popsány jsou i podporované režimy procesorů s architekturou RISC–V. Ty jsou důležité i pro framework, který je cílem této práce, neboť různé režimy kladou na procesor různé požadavky, a tím pádem se podle režimu procesoru bude lišit i testovací sada, která bude frameworkem vybrána.

Druhá část se zabývá popisem pojmu testování, rozlišuje jejich druhy a popisem současných trendů v této oblasti. Důraz je kladen zejména na oblast nefunkčního testování, neboť testování kompatibility je zařazeno právě do této skupiny.

Třetí část se zabývá popisem testovacího frameworku Pytest, který je použit pro implementaci vlastního frameworku. Popisuje jednotlivé fáze testovacího procesu v rámci běhu a vysvětluje možnosti ovlivňování běhu. Dále jsou zde popsány základní stavební kameny, které lze v Pytest použít pro co nejjednodušší implementaci testů.

Poslední část práce se zabývá návrhem, implementací a testováním frameworku pro spouštění testů kompatibility, jehož vytvoření je cílem této práce. V návrhu jsou specifikovány cíle frameworku a proveden rozbor požadavků, dle kterých je následný vývoj řízen. Jsou v něm navržené základní komponenty frameworku tak, aby bylo budoucí rozšiřovaní jednoduché. V implementační částí je pak popsán způsob řešení jednotlivých částí. Detailněji jsou popsány některé vybrané části, protože jde o složitější logiku, která nemusí být na první pohled zřejmá. V poslední fázi je provedeno experimentování na několika modelech procesoru RISC–V a jsou vyhodnoceny výsledky.

Kapitola 2

RISC-V

RISC-V je průmyslový standard pro specifikaci instrukční sady (ISA[1]). V současnosti způsobuje velký rozruch na poli trhu s procesory, neboť poskytuje univerzální instrukční sadu. Zájem o tuto architekturu projevily i nadnárodní společnosti, které mají zájem posouvat tento standard dále. Mezi tyto společnosti patří Google, IBM, NVIDIA, Qualcomm a jiné. Primárním cílem je vytvoření zcela otevřeného standardu instrukční sady, který bude dostupný jak na akademické půdě, tak v průmyslovém odvětví. Myšlenkou otevřenosti se tak podobá architekturám, které existují již delší dobu, např. OpenRisc, Amber (kompatibilní s čipy od ARM) nebo Zet (kompatibilní s x86). Výjimečnost této architektury spočívá ve faktu, že není navržena pro jedinou cílovou oblast procesorů – RISC-V se snaží cílit na všechny platformy, od malých vestavěných systémů, přes mobilní telefony, až po osobní počítače či dokonce serverové systémy [3, s. 2].

2.1 Vývoj architektury

Projekt RISC-V vznikl v roce 2010 na americké univerzitě University of California, Berkeley za účelem výuky a výzkumu procesorových architektur, avšak tato architektura přejímá některé své vlastnosti z dávnější historie. Zcela prvním předchůdcem byl vektorový procesor T0, jenž měl instrukční sadu založenou na instrukční sadě MIPS-II. Šlo o stroj, který podporoval privilegovaný režim. Hlavními návrháři tohoto procesoru byli Krste Asanović, Brian Kingsbury, Bertrand Irrisou a David Johnson.

Následně v roce 2000 byl na univerzitě MIT zahájen projekt Scale, který si zakládal na architektuře T0, avšak upustil od podpory opožděných skoků z architektury MIPS[2]. Od roku 2002, kdy spojením architektury MIPS a zjednodušené verze Scale vznikla architektura SMIPS, byl na MIT otevřen obor pro výuku technologií polovodičových integrovaných obvodů s velmi vysokou mírou integrace (angl. VLSI).

Na základě těchto předchozích projektů byl v létě 2010 zahájen projekt RISC-V. Prvotní verze 32-bitové instrukční sady byla použita pro výuku kurzu VLSI na univerzitě v Berkeley.

V současnosti bylo již vytvořeno několik implementací architektury RISC-V včetně zavedení jejich výroby. Jejich výčet lze vyčíst z tabulky 2.1.

První čip s RISC-V byl implementován v jazyce Verilog a vyroben za použití 28nm procesu v roce 2011. Jednalo se o dvě 64-bitová procesorová jádra. Prvním byl procesor, který dostal přezdívku "TrainWreck" kvůli velmi krátké době návrhu. Měl implementované

[1] Instruction Set Architecture

[2] Microprocessor without Interlocked Pipeline Stages

Název	Zahájení výroby	Výrobní proces	ISA
Raven-1	29. 5. 2011	ST 28nm FDSOI	RV64G1_Xhwacha1
EOS14	1. 4. 2012	IBM 45nm SOI	RV64G1p1_Xhwacha2
EOS16	17. 8. 2012	IBM 45nm SOI	RV64G1p1_Xhwacha2
Raven-2	22. 8. 2012	ST 28nm FDSOI	RV64G1p1_Xhwacha2
EOS18	6. 2. 2013	IBM 45nm SOI	RV64G1p1_Xhwacha2
EOS20	3. 7. 2013	IBM 45nm SOI	RV64G1p99_Xhwacha2
Raven-3	26. 9. 2013	ST 28nm FDSOI	RV64G1p99_Xhwacha2
EOS20	7. 3. 2014	IBM 45nm SOI	RV64G1p9999_Xhwacha3

Tabulka 2.1: Vyrobené testovací čipy

obvody, které umožňovaly detekci chyb v datech (angl. *error-detecting flip-flops*). Druhý z těchto procesorů měl připojenou 64-bitovou jednotku umožňující vektorové výpočty s čísly v plovoucí desetinné čárce.

Následně byla vyvinuta mikroarchitektura s instrukční sadou RV64 implementovaná v jazyce Chisel – nově vzniklým jazykem pro návrh hardware, vytvořeným na UC Berkeley. Její označení bylo Rocket a je i nadále vyvíjena za účelem vzniku parametrizovatelného generátoru procesoru RISC-V.

Čipy EOS14–EOS22 podporovaly standard IEEE[3] pro 64–bitové jednotky umožňující práci s vektorovými čísly v plovoucí desetinné čárce. Čipy EOS16–EOS22 obsahovaly dvě procesorová jádra s podporou koherentní urychlovací pamětí (angl. *cache*). Tyto procesory úspěšně překročily taktovací frekvenci 1,25 GHz.

Následovala implementace instrukčního simulátoru pojmenovaného *Spike* po zlatém bodci, který symbolizoval dokončení mezikontinentální vlakové tratě ve Spojených státech amerických [3, s. 127]. Model byl implementován v jazyce C++Andrewem Watermanem a Yunsupem Leem a měl sloužit jako referenční model pro vývoj. Spike je dostupný pod BSD licencí.

Směr vývoje standardu zastřešuje nezisková organizace RISC-V Foundation, ve které jsou zastoupeny i výrazné společnosti na trhu s hardwarovými produkty, např. NVIDIA, Google, AMD, Samsung a jiní.

2.2 ISA

Základem každé instrukční sady standardu RISC-V je soubor celočíselných instrukcí, které musejí být implementovány vždy. Ten obsahuje takové operace, které jsou nezbytné pro práci běžných nástrojů, tzv. toolchainu, kam patří kompilátor, assembler, linker a další nástroje. V současnosti jsou součástí standardu tři varianty celočíselné instrukční sady rozdělené podle šířky registrů a velikosti adresového prostoru – 32-bitová (RV32I), 64-bitová (RV64I) a 128-bitová (RV128I). Poslední z uvedených variant ještě není stabilní kvůli možným změnám využívání adresového prostoru, které přinesou 128-bitové architektury v budoucnosti.

Architektura RISC–V je navržena tak, aby umožňovala rozsáhlé rozšiřování a specializaci. Sada základních celočíselných instrukcí může být rozšířena o jedno či více volitelných rozšíření, ale tato základní sada nesmí být nikdy nadefinována jiným způsobem. Instrukční sadu tedy lze libovolně rozšiřovat dle potřeb pomocí standardních nebo nestandardních rozšíření. Standardní rozšíření by měla být obecně použitelná a neměla by

[3] Institute of Electrical and Electronics Engineers

kolidovat s ostatními standardními rozšířeními. Patří sem rozšíření pro násobení a dělení – (kódové označení "M", z angl. *multiply*), atomické instrukce (kódové označení "A", z angl. *atomic*), které popisuje způsob implementace instrukcí pro atomické čtení a atomický zápis z pamětí. Tyto operace jsou nezbytné pro meziprocesorovou komunikaci (např. při použití sdílené paměti) a synchronizaci. Dalším standardním rozšířením je rozšíření pro podporu čísel s plovoucí desetinnou čárkou v *single-precision*[4] (kódové označení "F", z angl. *floating point*) i *double-precision*[5] (kódové označení "D"). Všechna tato rozšíření s celočíselnou instrukční sadou dohromady, tedy "IMAFD", tvoří instrukční sadu s kódovým označením "G"(z angl. general–purpose), kterou podporují oficiální nástroje RISC-V. Mezi další definovaná standardní rozšíření patří:

- Q (*Quad precision*) – rozšíření pro čtyřnásobnou přesnost v plovoucí desetinné čárce (128–bitové registry).
- L (*Decimal floating–point*) – rozšíření pro čísla s plovoucí desetinnou čárkou. Základem pro exponent je číslo 10.
- C (*Compressed*) – rozšíření pro komprimované instrukce s kratší délkou.
- B (*Bit manipulation*) – rozšíření pro bitové instrukce, např. posun, rotace aj.
- J (*Dynamically translated*) – rozšíření pro dynamicky překládané instrukce. Může najít využití u jazyků jako je Java nebo Javascript.
- T (*Transactional Memory*) – rozšíření pro transakční paměť.
- P (*Packed SIMD*) – rozšíření pro SIMD[6] instrukce zpracovávající více datových toků v jedné instrukci.
- V (*Vector*) – rozšíření pro vektorové instrukce.
- N (*User–level interrupts*) - rozšíření pro instrukce umožňující přerušení na uživatelské úrovni.

2.3 Režimy procesoru

Režim procesoru určuje, které operace může nějaký proces na procesoru provádět. Prováděný kód totiž nemusí být zcela důvěryhodný, tj. jeho cílem může být poškození systému nebo přistoupení do části paměti, která pro něj není určena. Z toho důvodu v procesorech existují režimy procesoru, které definují omezení na oprávnění běžných aplikací (procesů). RISC-V standard definuje celkem 3 režimy procesoru, které jsou dostatečné pro jakýkoliv systém [2, s. 3] .

Prvním a zároveň nejvyšším režimem procesoru je tzv. strojový režim (mód M, angl. *machine mode*), ve kterém není provádění instrukcí žádným způsobem omezeno. Kód prováděný v tomto režimu je považován za důvěryhodný, neboť má nízkoúrovňový přístup k procesoru. Tento mód je základní a musí být implementován vždy, tedy i v případě, že daná implementace procesoru podporuje jiné režimy procesoru.

[4] Čísla s plovoucí desetinnou čárkou s jednoduchou přesností
[5] Čísla s plovoucí desetinnou čárkou s dvojnásobou přesností
[6] Single instruction multiple data – jedna instrukce zpracovává více datových toků

Dalšími režimy, které jsou podporovány z hlediska RISC-V standardu jsou uživatelský mód (mód U, angl. *user mode*) a privilegovaný mód (mód S, angl. *supervisor mode*). Uživatelský mód slouží pro běh běžných uživatelských aplikací, které mohou být potenciálně nebezpečné a mohou obsahovat kód, jehož cílem je napadení systému. Z toho důvodu je o vrstvu výše nad uživatelským režimem ještě privilegovaný režim. V privilegovaném režimu typicky běží operační systémy. Ty jsou z hlediska procesoru považovány za důvěryhodný software. Jeho primárním úkolem je zabezpečit správný chod systému například tím, že zavádí virtuální paměťový prostor, který umožňuje oddělit adresové prostory jednotlivých aplikací a každá běžící aplikace v uživatelském režimu je tedy pro ostatní zcela transparentní. Podporované kombinace režimů procesorů jsou uvedeny v tabulce č. 2.2.

Počet režimů	**Podporované módy**	**Použití**
1	M	Jednoduché vestavěné systémy
2	M, U	Bezpečné vestavěné systémy
3	M, S, U	Systémy podporující běh unixových operačních systémů

Tabulka 2.2: Povolené kombinace režimů procesoru

2.4 RISC-V Foundation

RISC-V Foundation je nezisková organizace založená v roce 2015. Jejím úkolem je udávat směr vývoje standardu RISC-V, včetně modifikací instrukční architektury. Členové této organizace se podílejí na vývoji specifikace standardní RISC-V architektury a k tomu vázaného hardwarového a softwarového ekosystému. Organizace má v současné době přes 100 členů, mezi které patří i mezinárodní korporace jako například Google, NVidia, Qualcomm, Samsung, Western Digital a mnoho dalších.

Představenstvo společnosti je tvořeno tzv. správní radou (angl. *Board of Directors*) sestávající se ze 7 zástupců - Bluespec, Google, Microsemi, NVidia, NXP, Kalifornská univerzita v Berkley a Western Digital.

RISC-V Foundation pravidelně organizuje akce, na kterých jednotliví členové organizace představují aktuální a budoucí vývoj architektury RISC-V.

Kapitola 3

Vývoj hardwaru a testování

Tato kapitola se zabývá stručným popisem vývojového cyklu hardwarového systému, jehož základy jsou důležité pro pochopení procesu testování. Po stručném popisu vývojového cyklu je proveden hlubší rozbor fáze testování, které je z pohledu této práce nejzajímavější etapa. Důraz bude kladen na popis způsobů testování, zejména na testování kompatibility, neboť vytvoření systému pro tento typ testování je předmětem této práce. Součástí ověření správnosti vyvíjeného produktu je i fáze verifikace, avšak ta není v této práci podrobněji rozebrána.

3.1 Životní cyklus vývoje hardwaru

Životní cyklus vývoje hardwaru je časové období od přípravy specifikace požadavků až do ukončení vývoje produktu. Cyklus je rozdělen do několika fází, z nichž každá fáze má jasně definované vstupy a výstupy. Základními fázemi životního cyklu hardwaru jsou:

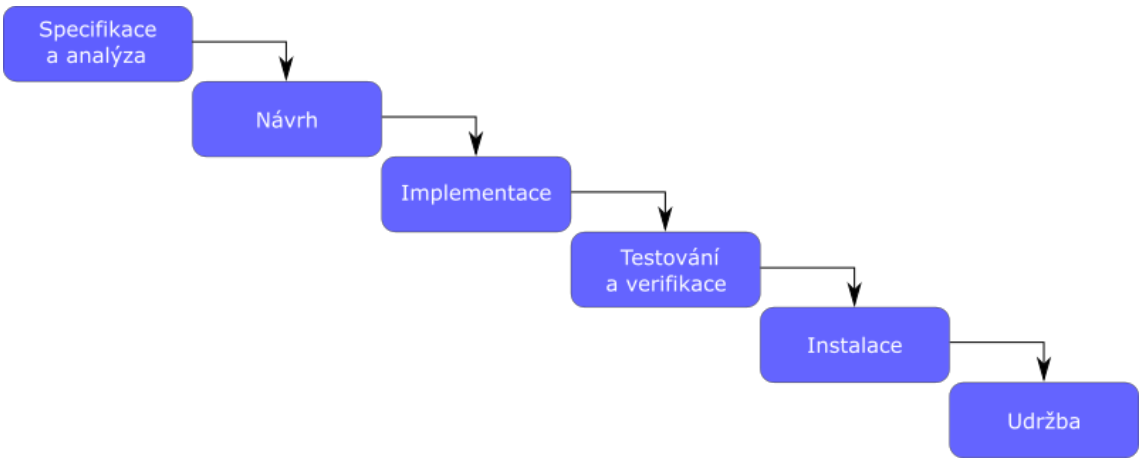
- Specifikace a analýza požadavků
- Návrh
- Implementace
- Testování a verifikace
- Instalace
- Údržba

U výše popsaných fází si lze všimnout podobnosti s životním cyklem softwarového systému, avšak samotná realizace těchto fází je odlišná. Největší rozdíl je ten, že hardwarový produkt je navrhován, zatímco softwarový produkt je vyvíjen. Ačkoliv se na první pohled jedná o pouhou terminologii, není tomu tak. Návrh hardwaru je ukončen ve chvíli, kdy je doručen zákazníkovi. Od tohoto okamžiku nemá návrhář žádnou kontrolu nad vydaným produktem. Na druhou stranu vývoj softwarového produktu je iterativní proces, u kterého lze produkt neustále vylepšovat pomocí záplat. Vývoj nové verze hardwaru je velmi drahý a pomalý proces oproti nové verzi softwaru, jehož aktualizace je levná a rychlá. Jako příklad rozdílu lze použít chyby pojmenované Meltdown a Spectre, které se na konci roku 2017 objevily v procesorech společností Intel a ARM [7]. Jde o bezpečnostní chyby, kdy může běžný uživatelský proces přistupovat k chráněným datům jádra systému, díky provádění

instrukcí Out-of-Order[1]. Jelikož jsou procesory fyzické produkty, nelze tuto chybu opravit přímo v procesoru, avšak lze ji částečně opravit aktualizací operačního systému, který na daném procesoru běží. Chybu ve fyzických zařízeních je však možné odstranit až v dalších verzích procesorů.

V první fázi – specifikace a analýza požadavků – probíhá prvotní komunikace se zákazníkem. Jsou specifikovány požadavky na produkt a požadavky na návrh. Cílem je i vytvoření dokumentu zvaného Master Test Plan, který jasně definuje postup při testování produktu, například obsahuje časové naplánování testovacího procesu, jaké vlastnosti produktu mají a nemají být testovány, prostředí pro provádění testů, kritéria, kdy je test označen za procházející nebo naopak selhávající, popisy testovacích případů, zodpovědnosti lidí a další informace.

Dále je vytvořen dokument funkční specifikace, který rozšiřuje požadavky popsané ve specifikaci produktu s takovou mírou detailu, aby podle nich mohly být navrženy elektronické obvody, 3D modely a další náležitosti týkající se návrhu. Obsahuje tedy specifikaci komunikačního rozhraní, požadované vstupy a výstupy, propustnost, požadavky na spotřebu, způsob ošetřování chybových stavů, požadavky na velikost paměti, prostředí a jiné. Následně je vytvářen elektronický a mechanický návrh. Součástí elektronického návrhu jsou blokové diagramy a nákresy schémat na základě funkční specifikace. Mechanický model zobrazuje 3D modely a 2D nákresy výsledného produktu.

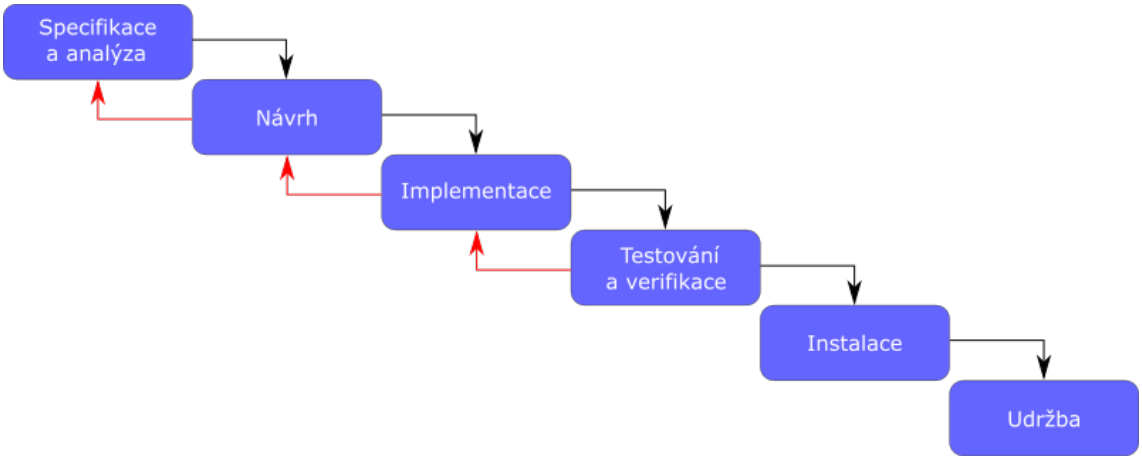


Obrázek 3.1: Vodopádový model.

Jedním z možných přístupů při vývoji je využití návrhového modelu vodopád, jehož diagram je zobrazen na obrázku 3.1. V jeho čisté podobě jde o sekvenční vývojový proces, ve kterém lze na další fázi přejít až v okamžiku, kdy je zcela dokončena fáze předchozí [10]. Při vývoji reálných produktů ale často dochází ke změnám specifikace či návrhu i v době, kdy je rozpracovaná fáze implementace. Z toho důvodů se používá upravený vodopádový model s názvem Roycův vodopádový model, který je znázorněn na obrázku 3.2. Ten umožňuje návrat k předchozí fázi vývoje produktu.

Dalšími používanými modely jsou například prototypování nebo inkrementální (agilní) vývoj. Základem prototypování je vytvoření částečně funkční verze produktu, který se následně poskytne zákazníkovi na získání zpětné vazby. Zpětná vazba slouží k ověření, zda produkt splňuje očekávání zákazníka. Inkrementální vývoj je agilní metodikou, která se za-

[1]Out-of-Order – vykonání instrukcí v jiném pořadí než v jakém jsou zapsané v programu za účelem minimalizace plýtvání hodinovými takty.



Obrázek 3.2: Roycův vodopádový model.

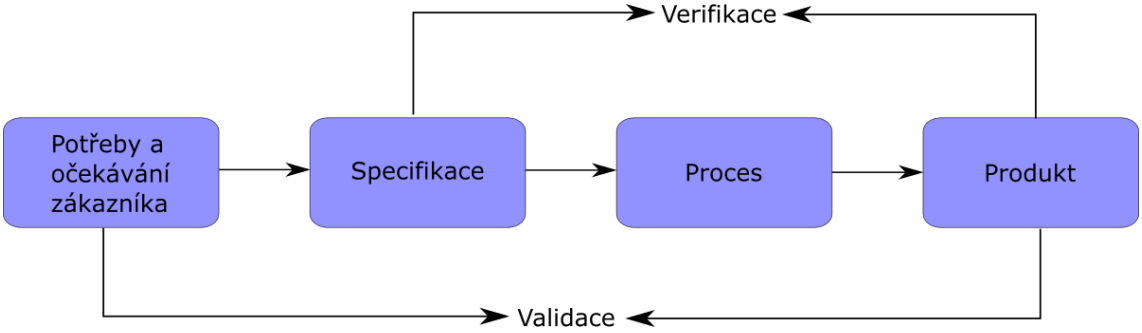
měřuje na flexibilitu vývoje tím způsobem, že oproti předchozím životním cyklům zkracuje délku jednotlivých fází typicky na jednotky týdnů, avšak celý cyklus se několikrát opakuje. Pro tento přístup je velmi důležitá častá komunikace se zákazníkem, který sleduje vývoj produktu.

3.2 Testování

Testování je klíčovým faktorem během vývoje jakéhokoliv produktu. Jde o proces, který ověřuje, že výstupy testovaného produktu jsou v souladu s očekávanými výstupy. První testování začíná jakmile existuje nějaká první verze modelu či prototypu produktu. Nejdůležitějšími úkoly testování jsou:

- Předejít defektům produktu v brzké fázi vývoje. Tento úkol je nejdůležitějším úkolem testování, neboť minimalizuje riziko výskytu defektu v budoucnosti, což zlepšuje jeho udržovatelnost. Z dlouhodobého hlediska také pomáhá snížit celkovou dobu vývoje produktu, a tím i cenu vývoje.
- Detekovat a odstranit chyby dříve, než je odhalí zákazník.
- Zajištění, že produkt splňuje požadavky, potřeby a očekávání zákazníka. Tento požadavek zajišťuje validace a verifikace produktu.

Validace a verifikace jsou procesy, které jsou součástí testování. Verifikace zajišťuje, že produkt odpovídá jeho specifikaci, čili se soustředí na otázku "Je produkt vytvářen správně?". Provádí ji tým, který požadovaný produkt vyvíjí a obvykle předchází validaci. Spadá sem například funkční a formální verifikace, jednotkové testování (angl. *unit testing*) a testování integrace (angl. *integration testing*). Validace na druhou stranu odpovídá na otázku "Je vytvářen správný produkt?". K validaci dochází až u finálního produktu a provádí ji zákazník, který se rozhoduje, zda produkt splňuje jeho požadavky a očekávání. V této fázi se používá např. akceptační testování (angl. *acceptance testing*) nebo testování použitelnosti (angl. *usability testing*). Názorná ukázka validace a verifikace je zobrazena na obr. 3.3.



Obrázek 3.3: Validace a verifikace.

3.3 Typy testování

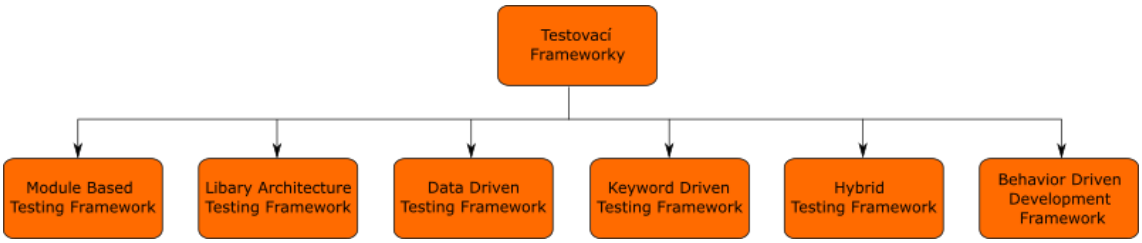
Testování lze rozdělit do skupin podle přístupu k testovacímu procesu, na funkční a nefunkční. Tomuto dělení bude věnována speciální podkapitola. Dalším rozdělením je statické a dynamické testování, kdy u statického testování není kód spuštěn, nýbrž jde o manuální kontrolu kódu, dokumentů apod. Provádí se zejména při vytváření dokumentace a prvotní fázi implementace. Primárním účelem je tedy prevence chyb. Naproti tomu dynamické testování je specifické tím, že dochází ke spouštění samotného kódu, tedy i k aplikaci vstupů. Hlavním úkolem je odhalit chyby způsobené za běhu. Velmi často jsou využívány nástroje, které umožňují pozastavení programu a krokovat jeho provádění – ladící programy, angl. *debuggery*. Dalším způsobem, kterým lze rozdělit testování na dvě skupiny, je použití metody černé nebo bílé skříňky. U prvního typu známe pouze rozhraní komponent, nikoliv způsob jejich vnitřní implementace. Naopak u metody bílé skříňky je známý veškerá vnitřní implementace testované komponenty.

V neposlední řadě lze rozdělit proces testování na manuální a automatické [1]. Během manuálního testování, jak název napovídá, jsou testy spouštěny ručně (manuálně) uživatelem bez použití jakýchkoliv nástrojů, či skriptů. Z toho důvodu je většinou časově náročnější, zvláště v případech, kdy jsou změny ve zdrojových souborech příliš časté. Velkou výhodu přináší například při testování uživatelského rozhraní, neboť automatické skripty neodhalí estetické nedostatky, které může odhalit pouze člověk. Další případ, kdy manuální testování vítězí nad automatickým vzniká v případě jednorázového otestování jisté funkcionality. Psaní automatických testů vyžaduje jisté časové úsilí a pro jednorázový běh testu může být manuální spuštění vhodnější. Na druhou stranu, tento způsob není vhodný, pokud vyžadujeme časté spouštění několika testů nebo testů s velkými daty. Také tím, že výstup kontroluje člověk, hrozí vyšší riziko přehlédnutí chyby. Automatické testování přináší oproti manuálnímu testování řadu výhod. Kvůli tomu, že využívá různé nástroje a automatické skripty, není během testování vyžadována žádná interakce s uživatelem. Je vhodný pro velké množství testů či velké množství vstupních dat. Výstupem automatického testování bývají hlášení o výsledcích testů, nejčastěji v formě logu.

3.4 Testovací frameworky

K automatickému testování se velmi často používají testovací frameworky, neboli spustitelné prostředí pro automatizované testy. Frameworky pomáhají uživatelům navrhovat a implementovat testy efektivním způsobem. Mezi výhodami, které použití testovacích frameworků přináší, jsou:

- Znovupoužitelnost kódu
- Maximální pokrytí řádků a kódu
- Nízká údržba
- Jednotné a přehledné zpracování výsledků



Obrázek 3.4: Typy testovacích frameworků.

Typy testovacích frameworků [6] jsou znázorněny na obr. č. 3.4. Jejich popis je následující:

- Modulární framework (*Modular Based Testing Framework*) – je založen na konceptu převzatého z objektově orientovaného programování – abstrakce. Již při implementaci je aplikace rozdělena do modulů, které jsou vzájemně nezávislé. Případná úprava jednoho modulu pak nemá vliv na ostatní moduly. Poté, co je aplikace rozdělena na dílčí moduly, je pro každý modul vytvořen testovací skript, který testuje funkcionalitu daného modulu. Na vrcholu existuje jeden testovací skript, který spouští testovací skripty jednotlivých modulů. Výhodou je to, že díky vytváření logicky uspořádaných modulů dochází k jednodušší údržbě testů. Frameworky tohoto typu jsou dobře škálovatelné. Nevýhodou je, že testovací data jsou v každém testovacím skriptu každého modulu. Tím pádem je v případě změny množiny dat nutné upravovat i testovací skript.

- Knihovní framework (*Library Architecture Testing Framework*) – ve svém základu je založen na výše uvedeném typu testovacích frameworků, avšak namísto vytváření zvláštních modulů je funkcionalita vyvíjené aplikace zapouzdřena uvnitř nějaké knihovní funkce. V rámci testu je pak volána stejná knihovní funkce a nemusí být implementována znovu v testovacím skriptu.

- Framework řízený daty (*Data Driven Testing Framework*) – u předchozích dvou typů byla testovací data vždy součástí testovacích skriptů. Frameworky tohoto typu ale oddělují data od testovací logiky takovým způsobem, že je jeden test možné spustit s různými množinami vstupních dat. Ta bývají obvykle uložena v externí databázi, např. ve formě souborů XML[2], CSV[3], textových souborů, tabulkových souborů apod. Hlavní výhodou tedy je to, že je znatelně snížen počet testovacích skriptů, je zvýšena flexibilita a udržovatelnost; při změně testovacích dat nemusí být editován testovací skript. Nevýhodou je náročnost návrhu takových testů a nutnost implementovat čtení dat z externí databáze.

[2] eXtensible Markup Language
[3] Comma-Separated Values

- Framework řízený klíčovými slovy (*Keyword Driven Testing Framework*) – rozšiřuje Data Driven framoworky tím, že součástí souborů externí databáze jsou i tzv. klíčová slova, která určují samotný průběh testu. Ke každé funkci testované aplikace je vytvořen seznam kroků, které jsou reprezentovány klíčovým slovem a akcí, kterou dané slovo představuje. V testovacím skriptu jsou pak vyjmenována klíčová slova, která mají být v daném testu vykonány. Tento přístup se používá zejména pro testování grafického uživatelského rozhraní (např. webové rozhraní).
- Hybridní framework (*Hybrid Testing Framework*) – kombinuje všechny výše uvedené varianty.
- Frameworky řízené chováním (*Behavior Driven Development Framework*) – testovací případy jsou zapsány ve formě přirozeného jazyka, který je dobře čitelný pro obchodní analytiky, vývojáře i testery. Tyto frameworky nevyžadují znalost programovacího jazyka. Příklad testu při použití frameworku řízeného chováním je znázorněn na příkladu 3.1.

```
Given the account is in credit

And the dispenser contains cash

When the customer requests cash

Then ensure the account is debited

And ensure cash is dispensed

And ensure the card is returned
```

Příklad 3.1: Ukázka testu ve frameworku řízeném chováním [5].

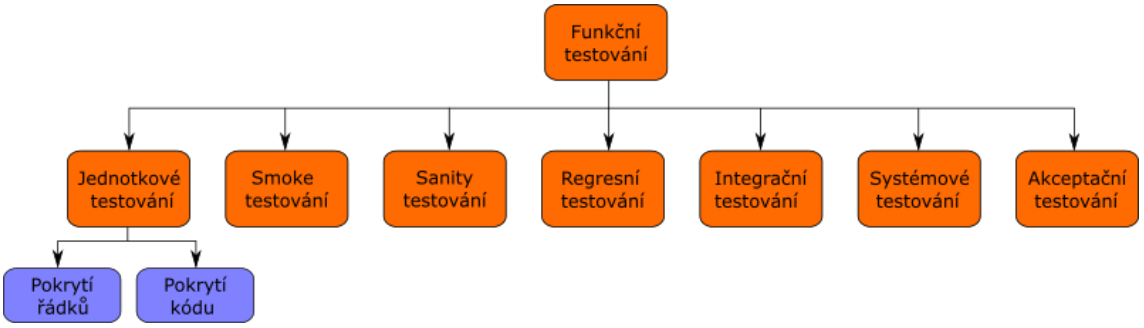
3.5 Funkční testování

Funkční testování (angl. *functional testing*) je typ testování, který se opírá o specifikaci požadavků a funkční specifikaci, která je vytvářena na počátku životního cyklu produktu. Jde o typ testování pomocí černé skříňky.

Zahrnuje kompletní otestování produktu a ověření souladu se specifikací. Testeři ověřují specifickou akci, či funkci. Lze využít jak manuální, tak automatické testy. Tento typ testování v sobě zahrnuje celkem 4 kroky:

1. Vytvoření testovacích dat na základě specifikace.
2. Zjištění očekávaných výstupů dle funkční specifikace.
3. Spuštění testů.
4. Porovnání reálných a očekávaných výstupů.

Přehled kategorií funkčního testování je zobrazen na obr. č. 3.5. Jejich popis je následující:



Obrázek 3.5: Kategorie funkčního testování.

- Jednotkové testování – jednotkové testování obvykle provádí vývojář, který mění kód a potřebuje ověřit, že se se zadanými parametry chová funkce (jednotka) očekávaným způsobem. Součástí jednotkového testování bývají obvykle i metriky pokrytí řádků a pokrytí kódu.
- Smoke testování – kategorie testování, která obvykle zahrnuje malý počet jednoduchých testů. Testuje tedy velmi kritické funkce. Cílem je ověřit, zda je produkt či systém připraven na další testování. Typicky je prováděno okamžitě po vytvoření spustitelné verze.
- Sanity testování – je určeno ke zjištění, zda byly opraveny drobné chyby z předchozí verze. Primárně tedy šetří čas a zdroje tím, že pokud sanity testy selžou, pak nejsou spuštěny časově náročné regresní testy.
- Regresní testování – regresní testování pomáhá udržovat stabilní verzi produktu při jeho změnách. Pokaždé když je vytvořena nová verze produktu, která přidává novou funkcionalitu a opravuje chyby starší verze, jsou spuštěny regresní testy, jejichž výsledek určuje, zda tato úprava neměla za následek vznik nové chyby. Při použití inkrementálního vývoje, jsou tyto testy spouštěny na konci každé iterace. Regresní testování obvykle zahrnuje velké množství krátkých i dlouhých testů, proto je časově náročné.
- Integrační testování – integrační testování je na vyšší úrovni než jednotkové. Ověřuje se bezchybná komunikace mezi jednotlivými komponentami, případně i s okolím produktu. Integrační testování už neprovádí vývojáři, nýbrž testovací tým. U menších projektů je obvykle vynecháváno z důvodu, že pokud implementace obsahuje chybu, pak se chyba projeví i na vyšší úrovni testování.
- Systémové testování – systémové testování je poslední fází testování na straně vývojového týmu, které následuje po ověření všech předchozích typů testování. Ověřují produkt z pohledu zákazníka. Podle testovacích scénářů se simulují různé kroky, které mohou v praxi nastat. Součástí této fáze jsou i nefunkční testy, které jsou popsány v další sekci.
- Akceptační testování – kategorie testů, které provádí zákazník na své straně. Probíhá až v případě, že všechny předchozí etapy testů proběhly bez větších nedostatků.

3.6 Nefunkční testování

Nefunkční testování (angl. *non–functional testing*) je typ testování, ve kterém není cílem zajistit správnou funkcionalitu systému, nýbrž získat přehled o vlastnostech produktu, například rychlost odezvy, čas provedení jisté operace, výkon, bezpečnost a další vlastnosti. Z tohoto důvodu je nutné, aby funkční testování předcházelo nefunkčnímu. Má velký vliv na spokojenost zákazníka a uživatele při využívání produktu.

Kategorií nefunkčního testování a metrik pro měření kvality produktu existuje celá řada [9]:

- Výkonnostní testování – výkonnostní testování je obecný název pro testy, které sledují, jak se systém chová. Sledovanými vlastnostmi jsou například doba odezvy, stabilita, škálovatelnosti, spolehlivost, spotřeba zdrojů a další. Různé výkonnostní testy pak informují o dílčí metrice.

- Zátěžové testování – technika, při které je pozorována reakce produktu na vysokou zátěž. Používá se v případech, kdy je vyžadována odolnost proti zátěži nebo se ověřuje, zda testovaný systém uloží data před pádem nebo vypíše chybovou hlášku.

- Testování dokumentace – jde o časově náročnou aktivitu, neboť jej nelze efektivně provádět automaticky. Nástroje umožňují zkontrolovat gramatiku, ale nikoliv nejednoznačnost nebo nekonzistenci textu.

- Testování kompatibility – typ testování, který ověřuje, zda vyvíjený produkt odpovídá předem definovanému standardu či nikoliv. V následující podkapitole je tento typ testování rozveden blíže.

- Testování spolehlivosti – typ testování, které ověřuje, zda vyvíjený produkt pracuje bez chyb po pevně daný časový úsek v daném prostředí. Jeho účelem je odhalení struktury opakujících se chyb, zjištění časového intervalu, ve kterém dochází k chybám a následná detekce příčiny a odstranění chyby. Běžnými ukazateli tohoto typu testování jsou střední doba poruchy (angl. *Mean Time To Failure*) a střední doba do obnovení (angl. *Mean Time To Restore*), jejichž součtem je metrika střední doba mezi poruchami (angl. *Mean Time Between Failures*). Čím vyšší tato hodnota je, tím je produkt spolehlivější.

3.7 Testy kompatibility

Testování kompatibility ověřuje, zda produkt nebo systém splňuje požadavky na standard, specifikaci, či regulaci. Tyto standardy a specifikace jsou velmi často definovány velkými nezávislými entitami, jako například Institut pro elektrotechnické a elektronické inženýrství (IEEE). Tento typ testování spadá pod nefunkční testování, neboť není ověřováno správné chování vůči specifikaci požadavků daného produktu, nýbrž vůči standardu organizace. Metodika testování je černá skříňka; to znamená, že jsou ověřovány pouze rozhraní produktu. Jinými slovy, pokud rozhraní obdrží požadavek, pak se testuje, zda toto rozhraní umožňuje požadavek přijmout a odpovědět správně.

V určitých situacích nemusí produkt odpovídat celému standardu, ale pouze jeho části. Toho se využívá zejména u velmi rozsáhlých standardů. V takových případech je pak zapotřebí definovat, které části jsou pro daný produkt důležité. K popisu těchto částí se využívají

tzv. klauzule kompatibility, které jsou určeny pro lepší porozumění částí standardu návrháři či vývojáři.

Některé produkty nevyžadují podporu celého standardu. Může také nastat případ, že daná implementace toto neumožňuje. V takovém případě se zavádí pojem profil a úroveň. Profil je taková podmnožina standardu, která je dostačující pro uspokojení požadavku kompatibility. Úroveň je pak vnořená podmnožina daného standardu. Typicky bývá úroveň 1 jádro standardu. Úroveň 2 pak zahrnuje vše co úroveň 1 a k tomu další funkcionalitu specifikovanou standardem. Takových zanoření může být až n, což zahrnuje celý standard [8].

3.8 Testovací frameworky v jazyce Python

Pro implementační část této práce je využit programovací jazyk Python. Tento jazyk byl vybrán, protože jde o velmi flexibilní programovací jazyk, což umožňuje rychlý vývoj produktů v tomto jazyce. Je také snadno čitelný, a proto může snižovat úsilí při čtení cizího kódu. Interpret jazyka Python je implementován pro několik operačních systémů, včetně systémů Windows a Linux. Python je tedy platformně nezávislý a stejný kód lze spouštět na různých operačních systémech.

V této podkapitole jsou stručně popsány používané testovací frameworky, které již v jazyce Python existují. Testovacích frameworků existuje v Pythonu celá řada, avšak valná většina z nich již není udržována, proto nejsou blíže prozkoumány.

Popsané frameworky umožňují definovat celkem 3 standardní fáze průběhu testovacího případu:

- Setup – nastavení prostředí pro test, např. vytvoření připojení na databázový server.
- Execution – spuštění testu, porovnání výstupu s očekávanou hodnotou.
- Teardown – fáze pro návrat prostředí do stavu před provedením testovacího případu.

Nejznámější a nejčastěji používanými testovacími frameworky jsou frameworky PyUnit a Pytest. Praktická část práce je implementována ve frameworku Pytest, proto je popisu frameworku věnována celá kapitola, kde je framework detailně popsán.

PyUnit je testovací framework v jazyce Python, který je součástí standardní distribuce. Je založen na konceptu frameworku JUnit, který slouží pro jednotkové testování v jazyce Java. Slouží pro jednoduchou údržbu a automatizaci testování.

Základními komponentami frameworku PyUnit jsou:

- Fixtura – reprezentuje prerekvizitu pro daný testovací případ. Například vytvoření dočasného souboru nebo složky, spuštění procesu pro server, a jiné. Jedna fixtura může být využívána jedním nebo vícero testy.
- Testovací případ – nejmenší jednotka testování. Porovnává reálné hodnoty s referenčními výstupy.
- Testovací sada – kolekce testovacích případů, testovacích sad nebo jejich kombinace. Slučuje testy, které mají vždy běžet společně.
- Testovací řadič – komponenta, jejíž zodpovědností je řídit samotný běh testů a generovat výstup pro uživatele. Může mít grafické rozhraní, textové rozhraní nebo vracet speciální návratovou hodnotu reprezentující výsledek testování.

Příklad jednoduchého testu implementovaného pro framework PyUnit je ukázán na příkladu č. 3.2. Z příkladu je zřejmé, že testovací případy jsou součástí tříd, které sdružují testy, které mají běžet společně. Tato třída musí být potomkem *unittest.TestCase*, která poskytuje rozhraní pro porovnávání reálných a referenčním hodnot, viz metoda *assertEqual*.

Spuštění testovacího procesu se provádí spuštěním skriptu, ve kterém jsou dané testy implementované.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

if __name__ == '__main__':
    unittest.main()
```

Příklad 3.2: Ukázka testu v PyUnit

Kapitola 4

Pytest

Pytest je pokročilý testovací framework, který se na rozdíl od PyUnit snaží minimalizovat používání aplikačního rozhraní pro psaní testů. V testovacích případech tedy lze využívat čistý Python bez jakéhokoliv využívání předdefinovaných funkcí a metod.

Pytest rozděluje testovací proces na následující fáze:

- Inicializace testování
- Kolekce testů
- Spouštění testů
- Vytvoření hlášení

Popisu každé z nich je věnována speciální sekce, neboť každá z fází je velmi rozsáhlá. Pytest uživateli dává možnost ovlivnit jakoukoliv ze čtyř výše uvedených fází. K tomu slouží tzv. hooky, což jsou běžné funkce, které jsou automaticky volány frameworkem v určité fázi provádění testovacího procesu. Pomocí hooku může uživatel manipulovat s testy na základě parametrů z příkazové řádku, parametrů testu, testovacího prostředí a dalších vlastností. Stejně jako framework PyUnit umožňuje uživateli vytvořit hierarchické fixtury. To znamená, že fixtura může pro správné provedení vyžadovat jinou fixturu.

Velkou výhodou je možnost rozšíření frameworku. Toho lze dosáhnout pomocí zásuvných modulů (pluginů), které mohou definovat své vlastní hooky, fixtury nebo pomocné metody a funkce [4].

4.1 Inicializace testování

Inicializace testování je první fází při běhu testovacího procesu. Během této fáze dochází ke zpracování argumentů z příkazové řádky, registraci zásuvných modulů a konfiguraci běhu. Následuje popis jednotlivých hooků, které jsou dostupné v této fázi.

```
def pytest_addhooks(pluginmanager)
```

Hook slouží k registraci vlastních hooků. To je vhodné v případě, že do Pytestu chceme zaintegrovat vlastní plugin, který využívá vlastní hooky. Registrace hooku se provádí pomocí tzv. Plugin manageru, který implementuje metodu pro přidaní hooku. Této metodě stačí pouze předat Python modul, ve kterém se hooky nacházejí a manager automaticky registruje všechny funkce z daného modulu jako hooky.

```
def pytest_plugin_registered(plugin, manager)
```

Hook je zavolán v případě, že byl registrován nový zásuvný modul. Je užitečný například v případě, kdy je potřeba vynutit to, aby určitý zásuvný modul nebyl použit, např. protože koliduje s jiným zásuvným modulem, který naopak má být použit. V tomto hooku tedy může nově registrovaný modul odregistrovat, a tím zamezit jeho použití.

```
def pytest_addoption(parser)
```

Hook slouží pro přidaní podporovaných argumentů z příkazové řádky. Parser využívá pro zpracování argumentů modul argparse, který patří do standardních modulů jazyka Python (je součástí standardní instalace Pythonu). U každého argumentu lze nastavit:

- jméno argumentu,
- akci, která je provedena při detekci argumentu,
- implicitní hodnotu,
- množinu povolených hodnot
- a další vlastnosti.

```
def pytest_configure(config)
```

Hook slouží pro konfiguraci běhu. Nejčastěji zde dochází ke zpracování hodnot argumentů z příkazové řádky, a pokud je to vhodné, tak uložení hodnoty jako atribut objektu Config. Config je globálně dostupný objekt, který obsahuje zpracované argumenty a metody pro práci s pluginy a jinými hooky – skrze tento objekt je možné vyvolat vlastní hook definovaný ve svém zásuvném modulu.

```
def pytest_cmdline_parse(pluginmanager, args)
```

Hook slouží pro parsování argumentů z příkazové řádky a inicializaci objektu Config.

```
def pytest_cmdline_main(config)
```

Hook definuje chování Pytestu po spuštění. Implicitní implementace provede inicializaci (*pytest_configure*) a zavolá hook *runtest_mainloop*, jehož úkolem je řízení testovacího procesu.

4.2 Kolekce testů

Během kolekce testů pytest nalezne a vygeneruje všechny testovací případy na základě prohledávání souborového systému a argumentů z příkazové řádky. Po vygenerování testovacích případů má uživatel možnost každý z testovacích případů modifikovat nebo i přeskočit. Hooky v této fázi jsou následující:

```
def pytest_collection(session)
```

Řídící hook pro fázi kolekce testů. Provádí volání ostatních hooků této fáze. Po dokončení začíná fáze spouštění testů.

`def pytest_ignore_collect(path, config)`

Hook vrací hodnotu *True*, pokud daná cesta (argument *path*) nemá být využita pro kolekci testů. Jinými slovy, odstraňuje danou cestu ze seznamu prohledávaných cest.

`def pytest_collect_directory(path, parent)`

Hook je zavolán těsně před průchodem adresáře, který se nachází v argumentu *path*.

`def pytest_collect_file(path, parent)`

Hook je zavolán těsně před analýzou souboru, který se nachází v argumentu *path*.

`def pytest_generate_tests(metafunc)`

Hook slouží pro dynamickou parametrizaci testů. Argument *metafunc* obsahuje metadata o testovacím případu, například jaké fixtury testovací případ využívá. Zároveň obsahuje hodnoty markerů, kterými byl testovací případ označkován.

`def pytest_collection_modifyitems(session, config, items)`

Hook, který umožňuje modifikaci testovacích případů předtím, než je zahájeno testování. Je možné například některé testovací případy vypnout nebo přeskládat pořadí jejich provádění.

`def pytest_collectstart(collector)`

Hook slouží pro výpis dodatečných informací do konzole v okamžiku započetí fáze kolekce.

`def pytest_itemcollected(item)`

Hook slouží pro výpis dodatečných informací do konzole v okamžiku vytvoření testovacího případu.

`def pytest_collectreport(report)`

Hook slouží pro výpis dodatečných informací do konzole těsně před skončením fáze kolekce.

`def pytest_deselected(items)`

Hook slouží pro výpis dodatečných informací do konzole v okamžiku, kdy byl test odstraněn ze seznamu testů k provedení. Test je možné odstranit pomocí argumentu *-k*, jehož hodnota specifikuje regulární výraz, jež je aplikován na unikátní identifikátor každého vytvořeného testovacího případu. Pokud identifikátor neodpovídá předpisu regulárního výrazu, pak je testovací případ ignorován.

4.3 Spouštění testů

Fáze spouštění testů je jádrem testovacího procesu, neboť zde probíhá příprava testovacích případů (volání fixtur, které jsou požadované testovacími případy) a samotné spouštění testovacích případů.

```
def pytest_runtestloop(session)
```

Hook je proveden po dokončení kolekce testů.

```
def pytest_runtest_protocol(item, nextitem)
```

Hook volá hooky pro přípravu (*setup*), provedení (*call*) a dokončení (*teardown*) testovacího případu, včetně zachytávání výjimek vyvolaných za běhu.

```
def pytest_runtest_setup(item)
```

Hook je zavolán před hookem *pytest_runtest_call*.

```
def pytest_runtest_call(item)
```

Hook provede tělo testovacího případu.

```
def pytest_runtest_teardown(item)
```

Hook je zavolán po hookem *pytest_runtest_call*.

4.4 Vytvoření hlášení

Fáze vytvoření hlášení je prováděna souběžně s ostatními fázemi kvůli tomu, aby byla podoba hlášení zcela pod vlivem uživatele. V předchozí podkapitole byly popsány hooky, které slouží na úpravu hlášení ve fázi kolekce. Nyní budou přiblíženy ostatní hooky, které jsou důležité pro tuto fázi.

```
def pytest_runtest_makereport(item, call)
```

Hook je volán po skončení testovacího případu v jakékoliv fázi (příprava, provedení, dokončení). Je vhodný pro detekci, ve které z těchto tří fází test spadl a lze zjistit, proč případně testovací případ neprošel – parametr *call* mimo jiné obsahuje i informace o vyvolané výjimce.

```
def pytest_runtest_logreport(report)
```

Hook je volán po skončení testovacího případu v jakékoliv fázi (příprava, provedení, dokončení). Je vhodný pro úpravu hlášení například tím způsobem, že jsou přidána metadata testu. Možným příkladem metadat je zachycený chybový výstup.

```
def pytest_report_header(config, startdir)
```

Hook je volán ve fázi inicializace (po *pytest_configure*) a umožňuje do konzole vypsat dodatečné informace před spuštěním testu.

```
def pytest_report_teststatus(report)
```

Hook je volán po skončení testovacího případu v jakékoliv fázi (příprava, provedení, dokončení). Slouží k modifikaci implicitního konzolového výpisu. Pokud tedy testovací případ neprojde, pak je možné uživatele informovat o typu chyby, která nastala během provádění. Například tak lze rozlišit mezi selháním testu z důvodu vypršení časového limitu testu nebo selháním z důvodu špatné funkcionality.

```
pytest_terminal_summary(terminalreporter, exitstatus)
```

Hook slouží pro přidání sekce k celkovému hlášení, které je vypisováno do konzole. Implicitně toto hlášení obsahuje počet testů, které prošly, selhaly a byly přeskočeny.

4.5 Fixtury

Fixtura je funkce, která má být provedena před samotným spuštěním testovacího případu. Reprezentuje tedy prerekvizitu testu. Existují tři způsoby, jak lze fixturu použít. Prvním je, že její jméno použijeme jako parametr testu. V tomto případě je možné z fixtury získat její návratovou hodnotu. Příklad použití je uveden v příkladu č. 4.1. Při spuštění Pytestu se před vykonáním těla *test_file* zavolá funkce *test_dir* a její výsledek je funkci *test_file* předán jako parametr.

```
import os
import pytest

@pytest.mark.fixture
def test_dir(request):
    return os.path.dirname(str(request.node.fspath))

def test_file(test_dir):
    ...
```

Příklad 4.1: Ukázka použití fixtury pomocí jména

Druhým způsobem použití fixtury je použití dekorátoru *usefixtures*. Ten umožňuje specifikovat výčet fixtur, na jejichž provedení je test závislý. Avšak tento způsob má tu nevýhodu, že na rozdíl od předchozího způsobu, nelze získat návratovou hodnotu fixtury. Příklad vhodného použití je demonstrován v příkladu č. 4.2, kde fixtura *server* spustí webový server a v průběhu testu je pak zajištěno, že spojení s webovým serverem bylo navázáno.

```
import pytest

@pytest.mark.fixture
def server(request):
    server = WebServer(request.config.url)
    server.start()

@pytest.mark.usefixtures("server")
def test_webservice(request):
    assert connect(request.config.url) is not None
```

Příklad 4.2: Ukázka použití fixtury pomocí usefixtures

Posledním způsobem je použití parametru fixtury *autouse*. Ta zaručí, že fixtura je volána automaticky pro každý testovací případ bez toho, aniž by musela být explicitně vyjmenována. V některých případech je toto použití vhodné, avšak při nevhodném použití může mít nežádoucí následky. Ukázka použití je zobrazena v příkladu č. 4.3. Fixtura *server* bude automaticky provedena pro každý testovací případ.

```
import pytest

@pytest.mark.fixture(autouse=True)
def server(request):
    server = WebServer(request.config.url)
    server.start()

def test_webservice(request):
    assert connect(request.config.url) is not None
    ...
```

Příklad 4.3: Ukázka použití fixtury pomocí autouse

Fixtury v Pytestu umožňují specifikovat jejich rozsah působnosti (angl. *scope*). Ten má vliv na počet volání dané fixtury. Implicitně je fixtura provedena před každým voláním testovacího případu, avšak ne vždy je takový přístup žádaný. U dříve uvedených ukázkách kódu 4.2 a 4.3 by bylo dostačující, kdyby se server spustil pouze jednou za celý běh testovacího procesu a nebylo by nutné jej startovat před každým testovacím případem. Pytest podporuje celkem čtyři rozsahy působnosti:

- *function* – fixtura se zavolá před každým testovacím případem. Implicitní hodnota.
- *class* – fixtura se zavolá jednou pro všechny testovací případy v jedné třídě.
- *module* – fixtura se zavolá jednou pro všechny testovací případy v jednom modulu.
- *session* – fixtura se zavolá jednou pro všechny testovací případy v rámci testovacího procesu.

Vhodnější použití fixtury *server* z předchozích příkladů by mohlo vypadat tak, jak je uvedeno v příkladu č. 4.4.

```
@pytest.mark.fixture(scope='session')
def server(request):
    server = WebServer(request.config.url)
    server.start()
```

Příklad 4.4: Ukázka použití rozsahu působnosti fixtury

4.6 Markery

Markery jsou speciální značky, kterými lze označovat testy. Na jejich základě pak může být selektována pouze podmnožina všech implementovaných testovacích případů. Markery navíc umožňují dynamickou parametrizaci testovacích případů, což znamená, že jeden testovací případ může být spuštěn vícekrát s různými parametry. Velice snadno tak lze například označit testy, které trvají krátkou dobu a tyto testy pouštět v rámci ladění testovacích

případů. Markery jsou použity tak, že je jejich názvem dekorována testovací funkce nebo dokonce celá třída. Pytest implicitně podporuje sadu markerů, avšak uživateli nebrání v tom, definovat si vlastní marker, včetně jeho sémantiky. Předdefinovanými markery jsou:

- skip – testovací případ je vždy přeskočen.
- skipif – testovací případ je přeskočen pouze za určité podmínky.
- xfail – u testovacího případu se očekává, že selže.
- parametrize – testovací případ je spuštěn vícekrát s různými parametry.

Pro definici vlastního markeru stačí pouze dekorovat testovací případ markerem s požadovaným názvem a Pytest nový marker automaticky zaregistruje. Markeru lze navíc předat libovolné argumenty, které mohou být ve fázi kolekce testů zpracovány a na jejich základě může být testovací případ parametrizován. Ukázka vlastního markeru je demonstrována na příkladu č. 4.5. Jak bylo zmíněno výše, dle markeru lze selektovat i testy, které mají být spuštěny. To lze z příkazové řádky provést pomocí argumentu *-m <nazev markeru>*.

```
import pytest

@pytest.mark.custom_marker(arg1, arg2)
def test_marker():
    ...
```

Příklad 4.5: Ukázka použití markeru

Kapitola 5

Návrh

Cílem praktické části této práce je vytvoření prostředí pro spouštění testů kompatibility pro modely procesorů s instrukční sadou RISC-V. Prostředí bude sloužit jako nástroj pro ověření souladu implementovaného modelu se zvolenými standardními rozšířeními architektury RISC-V. V této kapitole budou popsány požadavky na výsledné prostředí a návrh architektury prostředí.

5.1 Specifikace a cíle frameworku

Hlavními požadavky na výsledný testovací framework jsou následující:

- Snadná rozšiřitelnost
- Jednoduché rozhraní a konfigurace
- Rychlá integrace na uživatelských strojích
- Přehledný a srozumitelný výstup
- Systémová nezávislost

Testovací framework bude využívat objektově orientovaný přístup. To usnadní možnosti při následném rozšiřování díky využití vhodných návrhových vzorů při implementaci. Testovací framework bude rozčleněn do několika logicky uspořádaných modulů tak, aby přidávání nových komponent a funkcionality bylo co nejjednodušší. Novými komponentami se má na mysli například podpora pro různé implementace vstupních modelů. Základní verze frameworku by měla obsahovat minimálně podporu pro instrukční simulátor Spike, který je volně dostupný a slouží jako referenční model pro instrukční modely (angl. *instruction accurate*) RISC–V, které mají implementované podporované instrukce, avšak nezabývají se časovou složkou. Důraz bude kladen také na to, aby si mohl koncový uživatel snadno vytvořit svou vlastní verzi modelu pro framework, který bude abstrahovat jím implementovaný RISC-V model.

Framework by měl poskytovat jednoduché rozhraní pro přidávání nových testů ověřující kompatibilitu modelu se standardem RISC-V. Uživatel si při spuštění bude moci zvolit, která standardní rozšíření mají být otestována. Framework bude mít implementované jak grafické uživatelské rozhraní, tak textové uživatelské rozhraní. GUI[1] bude sloužit pro přehledné ovládání a zobrazování výsledků, zatímco pomocí textového rozhraní bude možné

[1] Graphical User Interface

spouštění frameworku automatizovat, a tím urychlit a zjednodušit testování na straně uživatele.

Dalším z požadavků je rychlé zaintegrování frameworku na uživatelově počítači. Je tedy nutné uchovávat aktuální verzi frameworku na veřejně dostupném místě, aby bylo možné jej začít používat kdykoliv. Jako vhodný kandidát se jeví balíkovací index pro python (angl. *Python Package Index*), který slouží jako softwarový repositář pro Python. Instalace balíčku je pak provedena spuštěním příkazu *pip*, který je součástí standardní instalace jazyka Python. Další nespornou výhodou je možnost automatického testování frameworku. Jelikož je balíček dostupný on-line, je možné využít nástroje průběžné integrace (angl. *continuous integration*), které si framework nainstalují, a poté spustí testy. Obvyklým řešením pro tyto účely bývá systém Travis. Ten navíc podporuje automatické publikování nejnovější procházející verze balíčku. Způsob distribuce frameworku ale bude muset zvolit organizace RISC–V Foundation, pro kterou je tento framework implementován.

Po skončení běhu testů kompatibility musí framework vytvořit přehledné a srozumitelné hlášení výsledků. Výstupem budou celkem dvě detailní hlášení ve formátu HTML a XML. HTML formát bude sloužit pro uživatele, neboť umožňuje uživatelsky přívětivější zobrazení. Na druhou stranu formát XML bude dostupný z toho důvodu, aby bylo možné výstupy strojově zpracovávat. Z tohoto pohledu je XML soubor vhodné řešení.

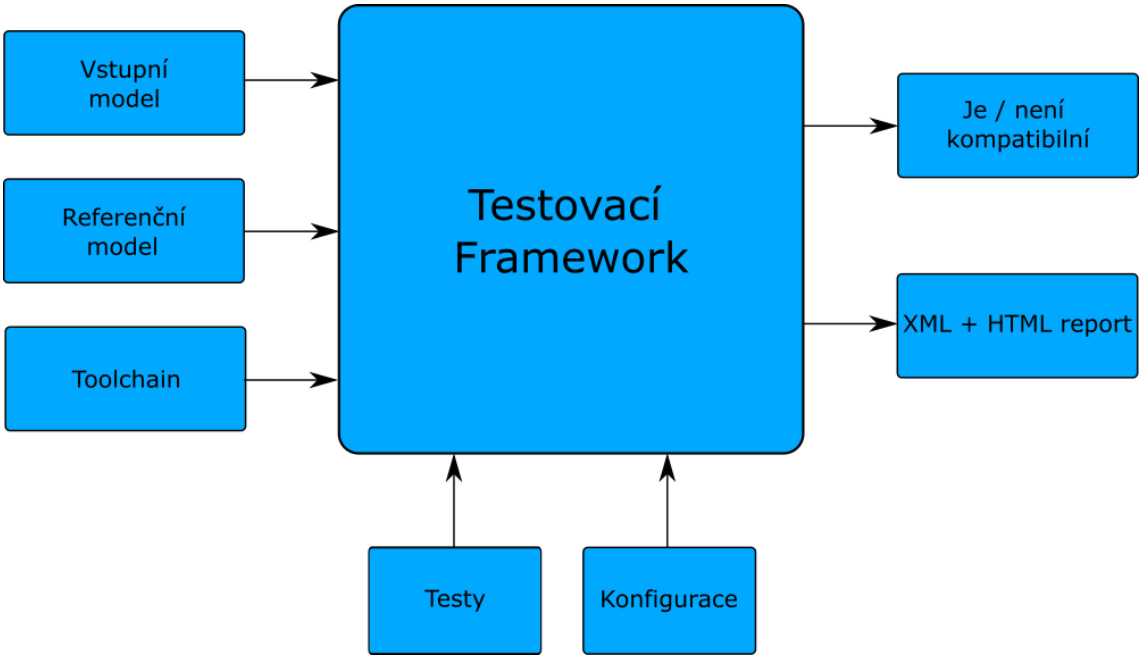
5.2 Návrh frameworku

Návrh frameworku z hlediska vstupně/výstupního rozhraní je zobrazen na obr.č. 5.1. Nezbytným vstupem je implementovaný model s architekturou instrukční sady RISC-V, který bude pomocí frameworku testován. Vstupní model může být buď čistý instrukční simulátor, nebo hardwarová implementace. Framework bude používat jako referenční model instrukční simulátor Spike, který je za referenční považován organizací RISC–V Foundation. Ten bude ve frameworku používán stejným způsobem jako testovaný vstupní model. Dále je potřeba sada nástrojů (angl. *toolchain*), které budou využity k překladu testovacích aplikací do spustitelné podoby. Součástí toolchainu jsou nástroje assembler, linker a pro zdrojové soubory v jazyce C je nezbytný i kompilátor jazyka C.

Vytváření zdrojových kódů pro testy není součástí této práce, proto jsou na obrázku zobrazující rozhraní frameworku znázorněny jako vstup. Tyto zdrojové kódy budou dodány organizací RISC-V Foundation, ve které existuje pracovní skupina vyhrazená na vytvoření a údržbu testů kompatibility. V rámci této skupiny pravidelně probíhají sezení, jejichž výstupem je snaha o definici podoby testů a metodiky pro určování požadavků na model, který by měl být kompatibilní se standardem RISC-V.

Testovací framework musí být konfigurovatelný. Testování kompatibility se standardem RISC-V tedy nemusí vždy ověřovat všechna standardní rozšíření popsaná v kapitole 2.2, ale je možné specifikovat pouze jejich podmnožinu. Díky tomu bude možné spouštět testy kompatibility již během vývoje modelu. Součástí konfigurace tedy bude například, zda má být testována 32, 64 nebo 128 bitová architektura instrukční sady a dále která standardní rozšíření daný model podporuje. Dle konfigurace bude automaticky selektována podmnožina testů kompatibility a ta bude následně spuštěna.

Po skončení testovacího procesu bude výstupem frameworku odpověď, zda je testovaný model kompatibilní s požadovanými rozšířeními standardu RISC-V a dále hlášení ve formátu HTML a XML. V každém hlášení bude seznam testů, které byly provedeny, včetně jejich výsledku a dalších informací jako např. které specifické rozšíření daný test ověřoval,



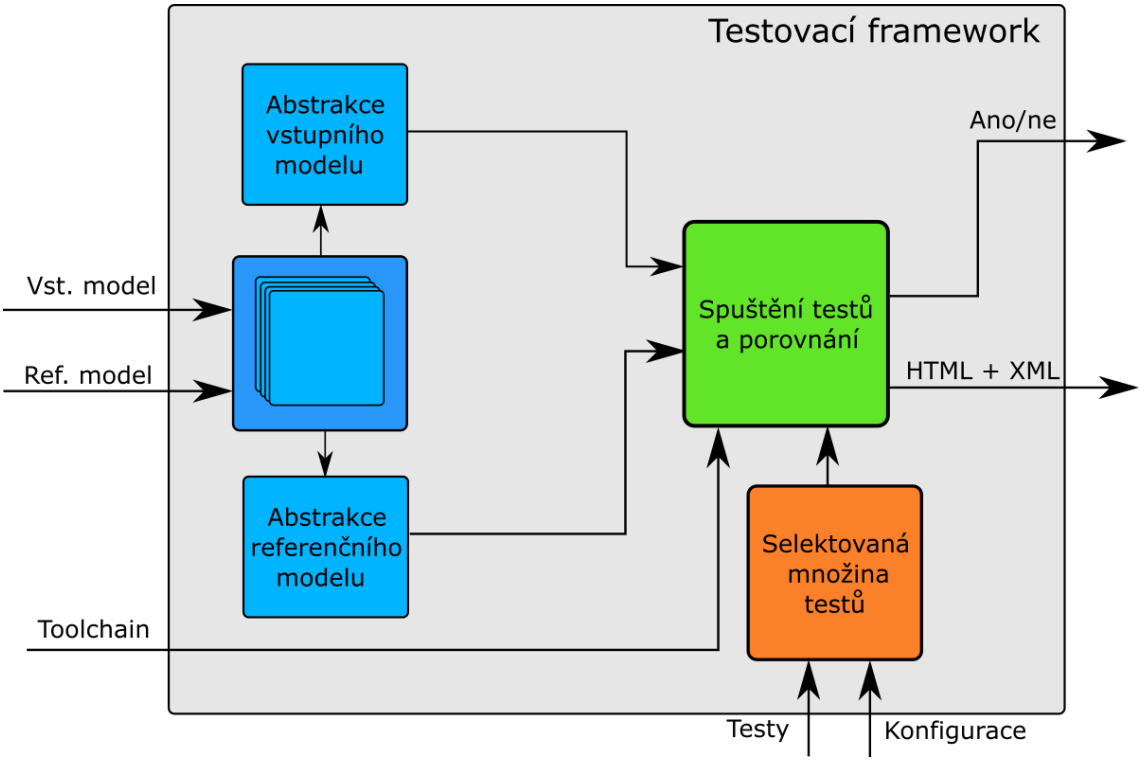
Obrázek 5.1: Rozhraní frameworku.

délka provádění apod. U testů, které skončily s výslednou chybou bude uvedeno zdůvodnění spadnutí testu.

Framework bude implementován takovým způsobem, aby byl systémově nezávislý. Bude tedy možné jej spustit pod vícero operačními systémy bez omezení jakékoliv funkcionality. Programovací jazyk Python je sám o sobě platformně nezávislý, čili tento požadavek je z technického hlediska splnitelný. V současné chvíli je ale problém ten, že referenční instrukční simulátor Spike je přeložitelný pouze pod systémem Linux. Nelze tedy očekávat funkcionalitu na systémech Windows do té doby, než bude implementace referenčního modelu pro tento systém existovat.

Vnitřní struktura frameworku je znázorněna na obr. č. 5.2. Framework bude obsahovat implementace abstrakcí modelů. Tyto abstraktní modely budou popisovat vlastnosti procesorových modelů, které budou využívány pro testování kompatibility. Budou tedy obsahovat konfiguraci modelu, která se mimo jiné skládá z následujících částí:

- Architektura instrukční sady – specifikuje bitovou šířku architekturálních registrů, které jsou definovány standardem a typ instrukční sady. Ten určuje, zda je model procesoru určen pro obecné použití nebo je optimalizován pro vestavěné systémy. Modely pro vestavěné systémy mají nižší počet registrů (16), zatímco ostatní mají 32 architekturálních registrů [3, s. 9-10].
- Podporovaná standardní rozšíření – seznam standardních rozšíření, která jsou podporována implementovaným modelem [3, s. i].
- Rozsah adresového prostoru – rozsah adresového prostoru má vliv na maximální velikost programu. Testy mohou v některých případech vyžadovat velké množství paměti, zejména, pokud testují různé režimy procesorů.
- Podporované typy přerušení – seznam výjimek, které model procesoru umí obsloužit. Například dělení nulou, nezarovnaný přístup do paměti a jiné [2, s. 35].



Obrázek 5.2: Vnitřní struktura frameworku.

- Podporované stavové registry – stavové registry uchovávají informace o stavu procesoru. Z hlediska standardu RISC-V je ale existence/implementace daných registrů odvozená od podporovaných vlastností procesoru. V procesoru tedy nemusí být vždy implementovány všechny stavové registry definované standardem. Z toho důvodu musí uživatel specifikovat, které registry jeho model implementuje [2, s. 9-12].

Hlavním cílem abstrakce modelů je jednotné rozhraní všech modelů, ať už instrukčního simulátoru nebo hardwarové implementace, aby bylo zajištěno co nejjednodušší použití těchto modelů.

Na základě dostupné množiny testů a konfigurace poskytnuté uživatelem bude vybrána podmnožina testů, která je relevantní pro proces testování. Výběr podmnožiny testů tak sníží celkovou dobu běhu testů a zamezí se zbytečnému spouštění testů, pro které neexistuje v procesoru podpora. Konfiguraci bude možné provést přes uživatelské rozhraní.

Spouštění testů bude pak probíhat tak, že každý zdrojový kód testu bude přeložen vstupním toolchainem, který je také konfigurovatelný, a jeho spustitelná podoba bude přivedena na vstup uživatelova i referenčního modelu. Po dokončení běhu testu budou výstupní hodnoty obou modelů porovnány a v případě shody bude test označen jako procházející; v opačném případě bude označen jako selhávající. Referenčním modelem mohou být i referenční soubory, které se budou nacházet u testů. V takovém případě bude test spuštěn pouze pro vstupní model a následně porovnán s referenčním výsledkem. V průběhu testování budou průběžně ukládány výsledky jednotlivých testů do paměti. Po dokončení testovacího procesu pak budou z těchto výsledků vygenerovány hlášení v dříve zmíněných formátech.

Kapitola 6

Implementace

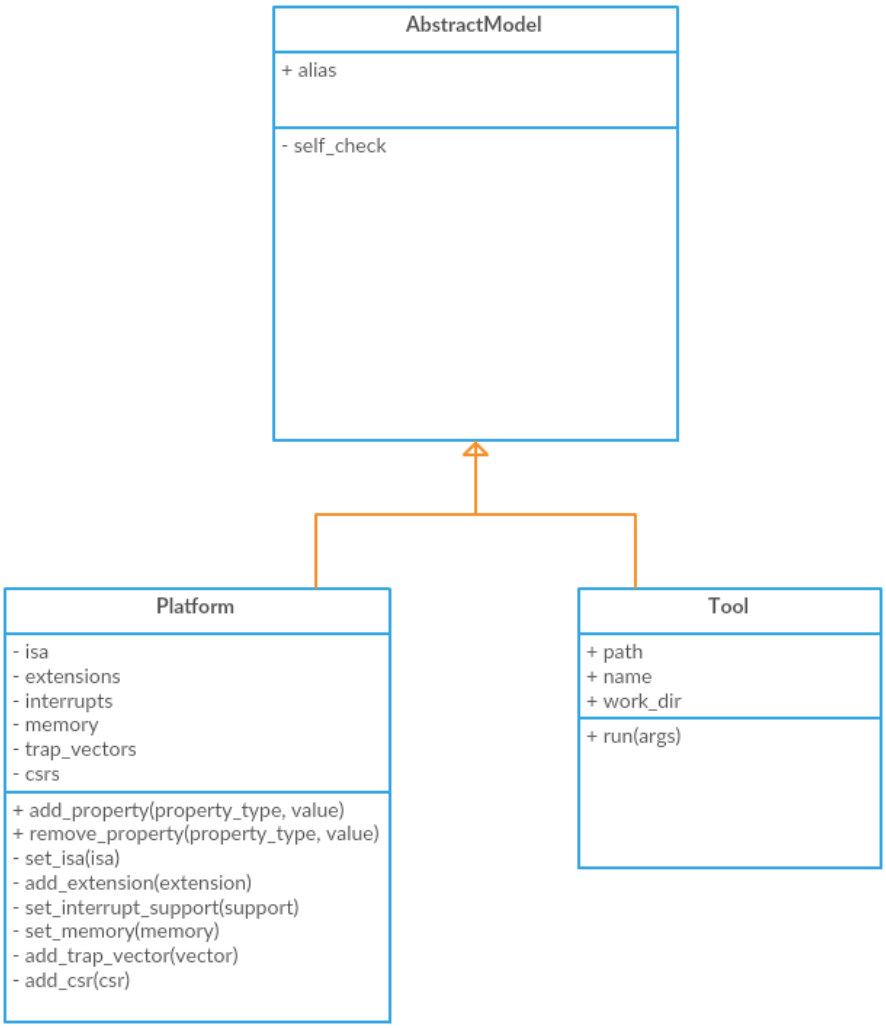
Tato kapitola se zabývá popisem prvků, jež je nutné vytvořit pro správnou funkcionalitu celého frameworku. Budou zde prakticky rozebrány části, které byly popsány v předchozí kapitole. První část kapitoly se věnuje popisu abstraktního modelu procesoru, který je využíván frameworkem. Dále je popsán generátor zásuvných modulů pro framework, jehož cílem je minimalizovat úsilí uživatele, který chce framework používat. Následně je popsána implementace testů ve frameworku a možnosti parametrizace jednotlivých testů. Je zde uveden i příklad funkčního testu. Ke konci kapitoly je popsáno uživatelské rozhraní a informace obsažené ve finálním hlášení.

6.1 Abstraktní model

Abstraktní model tvoří základní komponentu pro framework, jež je cílem této práce. Hlavní přínos tohoto prvku je ten, že definuje jednotné rozhraní pro práci s jakýmkoliv modelem procesoru, který bude testován vůči kompatibilitě se standardem RISC-V. To znamená, že ať je testovaný model procesoru implementovaný jakýmkoliv způsobem, je od frameworku odstíněna jeho vnitřní implementace, která pro testování není podstatná. Další výhodou je, že stejnou třídu lze využít jak pro testovaný model, tak i pro referenční model, kterým je v současné době simulátor Spike. Abstraktní model se skládá ze dvou částí, které jsou patrné z třídního diagramu na obr. č. 6.1.

První je tzv. platforma, která sdružuje vlastnosti procesoru, na základě kterých budou selektovány testy, které mají být spuštěny. Jinými slovy by se dala pojmenovat jako konfigurace procesorového jádra. Mezi tyto vlastnosti patří:

- Architektura instrukční sady - určuje, jaká základní instrukční sada má být použita při testování. Architektura instrukční sady udává šířku architekturálních instrukcí a také jejich počet – ten se liší podle toho, zda jde o model obecného procesoru nebo o model procesoru určeného pro vestavěné systémy (ty mají menší počet registrů kvůli úspoře adresového prostoru).
- Podporovaná standardní rozšíření – výčet podporovaných standardních rozšíření má zásadní vliv na fázi selektování testů, neboť optimalizuje dobu běhu tím, že v době běhu nejsou spuštěny testy pro nepodporovaná rozšíření – od těchto testů by se očekávalo, že automaticky neprojdou právě proto, že je daný model procesoru neimplementuje, a tím by testování trvalo zbytečně dlouhý čas.



Obrázek 6.1: Třídní diagram abstraktního modelu.

- Podporované režimy procesoru – režim procesoru má významný vliv na selekci testů, protože každý režim přidává speciální řídící instrukce, stavové registry a typy výjimek [2].
- Rozsah adresového prostoru – některé testy mohou vyžadovat vyšší velikost adresového prostoru, zejména pokud je podporován privilegovaný režim, který umožňuje běh unixových systémů na procesoru. Zároveň je nutné tuto informaci znát kvůli překladu zdrojových souborů. Velikost adresového prostoru má vliv na počáteční adresu kódu nebo umístění obsluh pro přerušení.
- Podpora výjimek a výčet podporovaných typů – jednoduché systémy nemusejí výjimky podporovat, proto je nutné tuto informaci o procesorů znát.
- Implementované stavové registry – stavové registry uchovávají informace o stavu procesoru. Jejich hodnoty jsou důležité v případě, že je vyvolána výjimka.
- Podpora nezarovnaných přístupů do paměti – pokud testovaný model procesoru umožňuje nezarovnané přístupy do paměti, pak musí existovat testy, které ověří, že tomu

tak skutečně je. Naopak, v případě, že je procesor nepodporuje, může být očekáváno vyvolání výjimky, která nezarovnaný přístup do paměti indikuje.

Platformu implementuje třída *Platform*, která zaobaluje výše popsané vlastnosti. Třída implementuje metody pro nastavování, přidávaní, odebírání a mazání těchto vlastností, přičemž před provedením každé takové operace je provedena kontrola, že je operace validní. Například, není možné do platformy přidat nějaké nestandardní rozšíření, neboť to není součástí RISC-V standardu, a tudíž není možné v rámci testovací sady ověřit, že je v souladu se standardem.

Druhou částí abstraktního modelu je třída *Tool*. Tato třída slouží jako obálka nad spustitelnými soubory a obecně usnadňuje práci s nimi. Umožňuje spouštět soubory stejným způsobem, jakým by probíhalo ruční spuštění z příkazové řádky. Navíc podporuje speciální argumenty, které dále mění prostředí spuštěného procesu, například:

- Spuštění procesu s jiným pracovním adresářem, než je pracovní adresář rodičovského procesu. Díky tomu není nutné před každým spuštěním podprocesu měnit pracovní adresář rodičovského procesu.
- Modifikace (přetížení) proměnných prostředí spouštěného procesu. Pokud nejsou proměnné prostředí nijak přetíženy, jsou použity proměnné prostředí rodičovského procesu.
- Nastavení časového limitu procesu. Pokud po nastavení časového limitu proces neskončil ani po dosažení časového limitu, pak je proces automaticky ukončen.
- Přesměrování (zachycení) standardního a standardního chybového procesu. To umožňuje po skončení procesu dále pracovat s výstupy procesu. Toho je využíváno zejména při detekci příčiny selhání testu.

Po skončení běhu procesu je k dispozici objekt reprezentující výsledek běhu. Ten obsahuje informace o návratovém kódu, příkaz spuštění a zachycený standardní a standardní chybový výstup. Tyto informace jsou dostačující k následné detekci pádu, v případě, že proces skončil s chybou.

6.2 Sada nástrojů

Framework ke své funkci potřebuje sadů nástrojů, která slouží k překladu zdrojových kódů testů. RISC–V Foundation poskytuje toolchain s překladačem *GCC*[1], který je upravený tak, aby podporoval architekturu RISC–V. Tento toolchain je potřeba, aby bylo možné vytvořit spustitelný soubor testu, který bude kompatibilní s referenčním simulátorem Spike. Framework poskytuje obálku pro tento překladač. Ta je implementována genericky, čili je možné ji použít pro jakýkoliv překladač podobný GCC.

Uživatel má možnost používat svůj vlastní překladač. V takovém případě ale musí sám implementovat metodu pro jeho spuštění. Pokud je jeho překladač odvozen od GCC, pak lze využít obálku, která je ve frameworku implementovaná. Referenční toolchain od organizace RISC–V Foundation je frameworku předán skrze proměnnou prostředí *RISCV*. Tato proměnná prostředí je standardně využívána při překladu RISC-V toolchainu, a proto se předpokládá, že ji uživatel nastavil správně. Pokud uživatel proměnnou nastavenou nemá

[1] GNU Compiler Collection

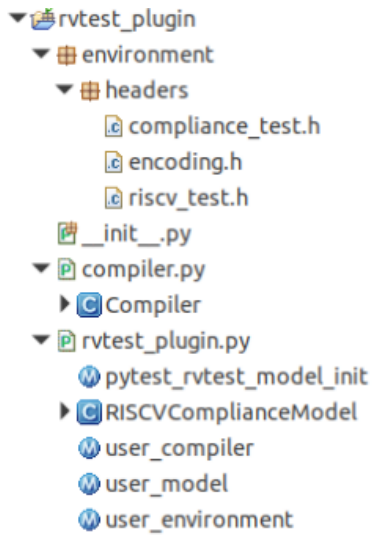
nebo neobsahuje cestu k RISC–V toolchainu, pak je o této skutečnosti informován a testovací proces je ukončen.

Framework nabízí automatické vytvoření RISC–V toolchainu. Uživatel tedy nemusí toolchain překládat ručně. Zdrojové soubory toolchainu jsou dostupné z veřejného repozitáře; framework tedy repozitář stáhne a nástroje přeloží do uživatelem specifikované cesty. Tuto cestu lze specifikovat buď při spuštění překladu z příkazové řádky pomocí parametru, nebo pokud má uživatel definovanou proměnnou prostředí RISCV, pak je použita ona. Pokud ani proměnná prostředí není nastavena, pak se implicitně použije pracovní adresář frameworku.

6.3 Zásuvný modul

Dříve popsaný abstraktní model je frameworku předán ve formě zásuvného modulu kompatibilním s testovacím frameworkem Pytest, který je využit pro implementaci této práce. Uživatel má možnost tento zásuvný modul implementovat sám, pokud má dostatečné znalosti jazyka Python a Pytestu. Avšak lze využít i možnost automatického vygenerování zásuvného modulu pomocí generátoru, který framework poskytuje. Uživatel tedy v generátoru specifikuje vlastnosti abstraktního modelu, které by měly odpovídat vlastnostem implementovaného modelu RISC-V. Následně je vygenerovaný zásuvný modul předán frameworku, ten jej načte a spustí odpovídající testy. Jedna z možných variant zásuvného modulu je zobrazena na obr. č. 6.2. Samotný obsah se může lišit dle vlastností abstraktního modelu a uživatelem specifikovaných vlastností.

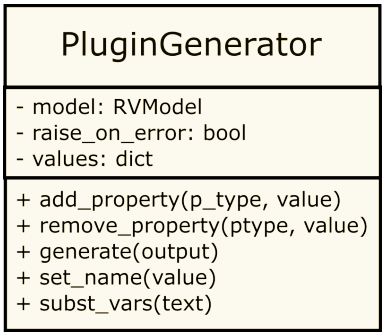
V kořenu adresářové struktury se nachází samotný modul rvtest_plugin.py, který je kompatibilní s frameworkem pytest. Ten implementuje abstraktní model a fixtury, které jsou použity při testování. Tyto fixtury popsány v sekci 6.6. Dále je zde volitelně modul implementující obálku nad kompilátorem, který bude použit pro překlad testů tak, aby testy bylo možné simulovat na modelu procesoru RISC-V, který chce uživatel otestovat. V případě, že uživatel nemá vlastní kompilátor, je použit překladač *riscv-gcc*, jehož zdrojové kódy jsou dostupné ve veřejných repositářích. Dále zásuvný modul obsahuje složku *environment*, která obsahuje hlavičkové soubory použité pro překlad testů. Tyto hlavičkové soubory obsahují definice maker, která jsou blíže popsána v podsekci 6.7.2.



Obrázek 6.2: Adresářová struktura zásuvného modulu.

6.3.1 Generátor zásuvných modulů

Generátor obsahuje sadu šablon, které jsou použity pro vygenerování modulů v jazyce Python. Do šablon jsou vložena data, která jsou specifikována uživatelem buď formou uživatelského rozhraní, nebo pomocí rozhraní, které může uživatel použít, když chce zásuvný modul vygenerovat ručně (tj. bez použití uživatelského rozhraní). Třída implementující generátor se jmenuje *PluginGenerator* a její UML diagram je zobrazen na obr. č. 6.3.



Obrázek 6.3: UML diagram generátoru

Instance třídy má uloženou instanci abstraktního modelu. Při nastavování nebo odebírání (metody *add_property* nebo *remove_property*) požadované vlastnosti abstraktního modelu je operace simulována na této instanci abstraktního modelu, která zajišťuje kontrolu správnosti hodnot. Pokud operace neproběhla správně, pak je zachycena výjimka vyvolaná abstraktním modelem a uživatel je skrze uživatelské rozhraní informován, že požadovaná vlastnost nemohla být přidána, a tudíž nebude ani ve vygenerovaném zásuvném modulu. Pokud ale operace proběhla v pořádku, pak je samotná hodnota uložena generátorem a následně použita při generování zásuvného modulu.

Přidávaní či odebírání vlastností modelu do generátoru je implementováno generickým způsobem, aby v případě rozšiřování abstraktního modelu nebyl nutný zásah do generátoru. Pokud ale ovšem přibude či bude odstraněna nějaká vlastnost, bude potřeba upravit šablonu zásuvného modulu tak, aby obsahovala pouze ty symboly, které budou skutečně nahrazovány.

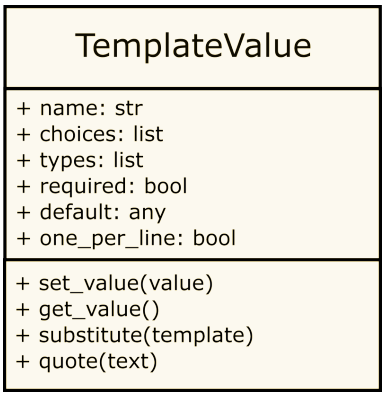
Ukázka zdrojového kódu pro vygenerování zásuvného modulu je zobrazena na obr. č. 6.1. Podporované typy vlastností jsou sdružené ve třídě výčtového typu (angl. *Enum*). Tím pádem je pro uživatele, který chce zásuvné moduly generovat ručně jednoduché zjistit, jaké všechny vlastnosti může u modelu nastavovat.

```
# Initialize generator
generator = PluginGenerator()
# Set ISA
generator.add_property(PlatformProperties.ISA, ISAS.RV32I)
# Set supported extensions
generator.add_property(PlatformProperties.EXTENSION, RiscVExtensions.A)
generator.add_property(PlatformProperties.EXTENSION, RiscVExtensions.M)
# Set memory size, program start address, data start address
generator.add_property(PlatformProperties.MEMORY, (0x8000, 0x100, 0x400))
# Generate plugin
generator.generate('rvtest_plugin')
```

Příklad 6.1: Použití generátoru

Generátor pro uchování hodnot, které budou následně vloženy do zásuvného modulu, využívá speciální třídu *TemplateValue*, jejíž UML diagram je na obr. č. 6.4. Ta zjednodušuje proces generování zásuvného modulu tím, že odděluje kód generování od kontrolování správnosti hodnot šablony. Pro každý symbol v šabloně existuje v generátoru instance třídy *TemplateValue*, která na něj klade určitá omezení, jejichž splnění je zodpovědností této třídy. Pro každý symbol lze specifikovat následující vlastnosti a omezení:

- *name* – název symbolu, za který bude v šabloně dosazena hodnota.
- *choices* – seznam povolených hodnot pro symbol. Pokud je seznam prázdný, pak nejsou na hodnotu kladena žádná omezení.
- *types* – povolené datové typy pro hodnotu. Omezení je vhodné zejména při použití výčtových tříd, kdy hodnota musí nutně být některým prvkem z dané výčtové třídy. Dále se tímto omezením specifikuje, zda může hodnota být i seznam hodnot nebo pouze jediná hodnota.
- *required* – příznak, který indikuje, zda musí pro daný symbol existovat nějaká hodnota, tedy zda je symbol povinný.
- *default* – implicitní hodnota symbolu, pokud není hodnota explicitně definována.
- *one_per_line* – příznak, který indikuje, zda má být každá hodnota symbolu (pokud jde o seznam hodnot) vygenerována na novém řádku. Typickým příkladem je symbol pro podporovaná standardní rozšíření. Pro tento symbol musí být každé rozšíření specifikováno samostatně, tj. nelze přidat více rozšíření v jednom volání metody generátoru *add_property*. Naproti tomu při nastavování velikostí pamětí se jedná o jednu hodnotu (trojici), viz příklad č. 6.1.



Obrázek 6.4: UML diagram TemplateValue

6.4 Selekce testů

Tato podkapitola obsahuje návrh možné realizace selekce testů, avšak konkretní implementaci musí být zvolena organizací RISC–V Foundation. V současné době není známo, jaké testy bude framework provádět, ani není definovaná podoba zdrojových kódů testů. Jedním ze způsobů, jak testy řadit je dle standardních rozšíření. Pro každé rozšíření bude existovat

složka obsahující testy pouze pro dané rozšíření. Skupiny testů v rámci jednoho rozšíření lze rozlišovat následujícími způsoby:

- Podle názvu souboru – název souboru bude obsahovat příznaky indikující, jakou vlastnost modelu daný test testuje. Mezi tyto vlastnosti může patřit specifická bitová šířka registrů, zachycení určité výjimky, přístup do nezarovnané paměti apod. Ve frameworku by se pak zdrojové kódy filtrovaly pomocí regulárních výrazů.
- Pomocí metadat ve zdrojovém kódu – zdrojové soubory mohou obsahovat metadata pro framework ve formě komentáře, který je ve fázi kolekce testů načte a dle metadat a vlastností modelu bude provedena selekce těch testů, které by na daném modelu měly projít.
- Kombinace obojího – některé příznaky by byly obsaženy v názvu souboru a některé v metadatech zdrojového kódu.

6.5 Kolekce testů

V rámci fáze kolekce testů je potřeba vyřadit testy, které testují vlastnosti, jež testovaný model nepodporuje; ne všechny vlastnosti jsou totiž povinné. Framework k tomuto filtrování nabízí vlastní markery, které slouží právě k tomuto účelu. Dle parametrů markerů jsou vyhledávány zdrojové kódy testů (assemblerovských souborů) a jsou dále filtrovány tak, aby odpovídaly specifikaci testovaného modelu. Následuje seznam implementovaných markerů:

```
pytest.mark.find_files(path='.', name='file', pattern=None, exclude=None)
```

Marker vyhledá rekurzivně všechny soubory specifikované argumentem *path*. Vyhledávání je prováděné relativně od kořenového adresáře obsahujícího testy. Následně jsou na všechny nalezené soubory aplikovány regulární výrazy z argumentů *pattern* a *exclude*, přičemž *pattern* je regulární výraz, který musí odpovídat názvu souboru, zatímco *exclude* je regulární výraz, který nesmí odpovídat názvu souboru. Argument *name* je název fixtury, pomocí kterého bude testovací případ parametrizován.

```
pytest.mark.architecture(isa=None, extensions=None, modes=None)
```

Marker slouží na selekcí testů pomocí architektury testovaného modelu. Je možné filtrovat na základě architektury instrukční sady, která ve svém názvu zahrnuje i bitovou šířku registrů. Lze tedy rozlišovat testy, které jsou určené pro různé bitové šířky instrukční sady. Druhý argument *extensions* slouží pro rozlišování standardních rozšíření, které daný test podporuje. Argument *modes* specifikuje režimy procesoru, které musí procesor podporovat. Mezi tyto módy patří *machine* mód, uživatelský mód a mód *supervisor*.

```
pytest.mark.memory(min_size=None, misaligned=False)
```

Marker filtruje testy pomocí vlastností paměti testovaného modelu. Argument *min_size* specifikuje minimální velikost paměti, kterou musí model mít. Tento filtr je vhodný zejména pro verzi instrukční sady RISC–V, která je určená pro vestavěné systémy. Ty mívají zpravidla menší velikost paměti než obecné procesory, určené například pro osobní počítače. Druhý argument *misaligned* určuje, zda test využívá nezarovnaný přístup do paměti. Pro procesory, které jej nepodporují nesmí být tyto testy spuštěny nebo musí být očekáváno vyvolání výjimky.

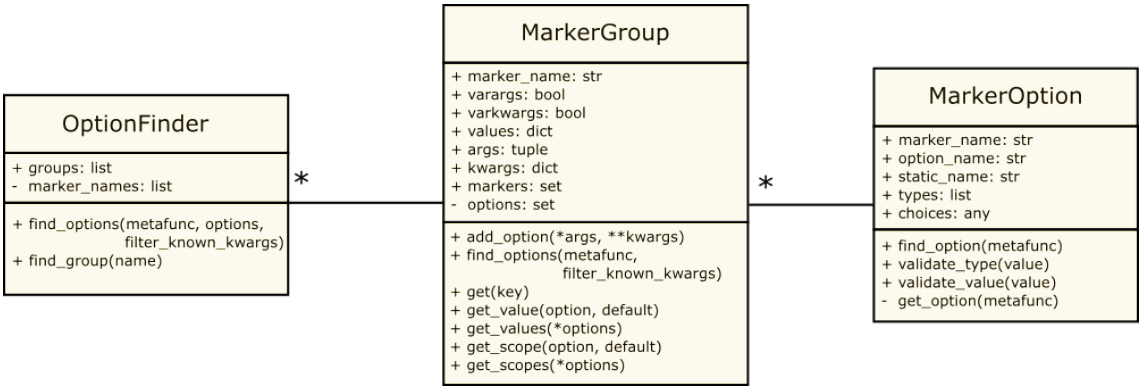
```
pytest.mark.exceptions(interrupt_support=False, *args)
```

Marker slouží na selekci testů na základě podpory určitých typů výjimek. Argument *interrupt_support* určuje, zda model podporuje alespoň některé typy přerušení (jednoduché procesory tuto podporu nemusí mít). Dále marker podporuje proměnný počet argumentů, které slouží pro specifikaci výjimek, jejichž podpora se od procesoru očekává.

```
pytest.mark.status_registers(*args)
```

Marker slouží pro filtrování pomocí implementovaných stavových registrů. Mohou existovat testy, které testují správné hodnoty v některých stavových registrech, proto je důležité spouštět testy pouze pro ty stavové registry, které procesor implementuje.

Filtrování testů na základě výše popsaných markerů se provádí ve fázi kolekce testů, konkrétně v hooku *pytest_collection_modifyitems*, který je popsán v sekci č. 4.2. V něm je získána hodnota markeru a porovnána s vlastnostmi abstraktního modelu, který vytvořil uživatel. Pro získání hodnot markerů je implementována třída *OptionFinder*. Ta využívá pomocné třídy *MarkerGroup* a *MarkerOption*, reprezentující samotnou hodnotu argumentu markeru. Třídní diagram je znázorněn na obrázku č. 6.5. Tento návrh umožňuje dynamicky kontrolovat argumenty markerů včetně ověření validity hodnot i jejich datových typů u každého testovacího případu.



Obrázek 6.5: Třídní diagram markerů

6.5.1 Třída MarkerGroup

Třída *MarkerGroup* reprezentuje marker, což je dekorátor, kterým lze označkovat testovací případ v Pytestu. Ten je definován názvem markeru a množinou validních argumentů markeru. Některé markery mohou obsahovat proměnný počet argumentů, proto musí i třída *MarkerGroup* tento případ užití podporovat. Význam atributů je následující:

- *marker_name* – jméno markeru, který je danou instancí reprezentován.
- *varargs*, *varkwargs* – určují, zda marker podporuje proměnné množství argumentů, resp. klíčových argumentů.
- *values* – slovník, jehož klíčem je název argumentu markeru a hodnotou je instance *MarkerOption*, která slouží k popisu jednoho argumentu markeru.

- *args* – seznam detekovaných proměnných argumentů, pokud jsou podporovány (*varargs = True*).
- *kwargs* – seznam detekovaných klíčových argumentů, pokud jsou podporovány (*varkwargs = True*).
- *markers* – množina instancí třídy *MarkerOption*, které náleží dané instanci *MarkerGroup*.
- *options* – Množina jmen argumentů markeru. Slouží pouze pro rychlejší vyhledávání podporovaných argumentů.

Přidání podpory pro nový argument do instance třídy *MarkerGroup* lze provést metodou *add_option*, které lze předat buď instanci třídy *MarkerOption* nebo argumenty, kterými lze novou instanci vytvořit. Opakovaným voláním metody jsou přidány všechny podporované argumenty.

Extrakce hodnot markeru je pak provedena pomocí metody *find_options*, jejímž povinným argumentem je objekt *metafunc*, který je popsán v sekci 4.2 při popisu hooku *pytest_generate_tests*. Skrze ni lze získat hodnoty markerů, kterými byl testovací případ označkován. Druhým argumentem je *filter_known_kwargs*, který pokud je nastaven, pak jsou z atributu *kwargs* odstraněny ty klíče, které jsou registrovány jako argumenty markeru. Jinak atribut *kwargs* obsahuje zcela všechny hodnoty markeru.

Nutno poznamenat, že při získávání hodnot markeru je vytvořená nová instance třídy *MarkerGroup* se stejnými hodnotami atributů a získané hodnoty markerů jsou uloženy v této nové instanci. Důvod je ten, že metoda *find_options* je opakovaně volána pro každý testovací případ. Tím pádem, kdyby byly hodnoty ukládány do původní instance, mohla by metoda v některých případech vracet špatné výsledky. Mezi tyto případy patří například ten, že některý testovací případ má definovaný nějaký argument markeru. U jiného testovacího případu ale tento argument nemusí být použit. Přesto by ale při získávání hodnoty byla navrácena hodnota z prvního testovacího případu.

6.5.2 Třída MarkerOption

Třída *MarkerOption* zaobaluje jeden argument nějakého markeru. Primárním úkolem je zajištění správného získání argumentu markeru, včetně typové kontroly a validity hodnoty argumentu. Pokud není hodnota validní, pak je vyvolána výjimka, která uživatele informuje o přesném typu chyby.

Pro získání hodnoty argumentu markeru je implementována metoda *find_option*. Ta nejprve získá hodnotu, u které ověří, že její datový typ odpovídá povoleným datovým typům, které jsou uloženy v atributu *types*. Skrze tento atribut lze i specifikovat, zda hodnota může být seznam, tj. může obsahovat více hodnot, nebo jde o jedno-hodnotový argument. Následně je zkontrolováno, jestli hodnota spadá do rozsahu povolených hodnot, které jsou dány atributem *choices*. Pokud hodnota obsahuje seznam, pak je kontrola provedena pro každý prvek tohoto seznamu. Implicitně není definováno žádné omezení na hodnotu ani datový typ.

Metoda markeru může být definována na vícero úrovních, tzv. *scopes*, hierarchickým způsobem. Nejvyšší prioritu má hodnota definovaná přímo v markeru. Pokud hodnota není nalezena, vyhledává se v třídních proměnných, pokud je testovací případ implementován v rámci třídy. Poslední úrovní pro hledání je úroveň modulu, tedy hodnota může být definována i jako globální proměnná. Tento přístup umožňuje zjednodušení při implementaci

testovacích případů, které mají společné vlastnosti. Hodnoty markerů lze definovat na globální úrovni, a tím pádem jsou stejné hodnoty použity pro všechny testovací případy, které se v daném modulu nacházejí. Hodnoty markerů tedy nemusí být definovaný pro každý testovací případ zvlášť. Název třídní nebo globální metody je uložen v atributu *static_name*, zatímco název argumentu v markeru (dekorátoru) je uložen v atributu *option_name*.

Na příkladu č. 6.2 je pro lepší pochopení ilustrován proces extrakce hodnoty.

```
# soubor conftest.py (definice hooku)

def pytest_generate_tests(metafunc):
    option = MarkerOption('custom_marker', 'option',
                          'CUSTOM_MARKER_OPTION')
    option.find_option(metafunc)

# soubor implementujici testovaci pripady

CUSTOM_MARKER_OPTION = False

class TestClass:

    @pytest.mark.custom_marker(option=True)
    def test_basic(self):
        ...

    def test_fast(self):
       ...

def test_slow():
   ...
```

Příklad 6.2: Získání hodnoty markeru

Hook *pytest_generate_tests* je spuštěn pro každý testovací případ, tedy *test_basic*, *test_fast* a *test_slow*. Získání hodnoty markeru *custom_marker* probíhá hierarchickým způsobem popsaným výše. Tedy nejprve je prozkoumán dekorátor funkce. Ten je definován u funkce *test_basic*, proto z metody *find_option* bude navrácena hodnota *True*. U ostatních dvou testovacích případů hodnota není definována ani v markeru, ani jako třídní atribut. Tudíž se prohledávání provede na úrovni modulu, kde je definovaná proměnná *CUSTOM_MARKER_OPTION*, jež odpovídá statickému jménu markeru a je tedy použita tato hodnota (*False*).

6.5.3 Třída OptionFinder

Metody třídy *OptionFinder* slouží jako aplikační rozhraní pro získávání hodnot markerů z testovacích případů. Odstiňuje od uživatele logiku spojenou s extrakcí hodnot a poskytuje jednoduchou metodu pro získání všech potřebných hodnot. Při vytváření instance této třídy je zapotřebí seznam instanci třídy *MarkerGroup*. Tímto si *OptionFinder* uloží podporované markery a je připraven na použití. Uživatel pro extrakci hodnoty markeru používá metodu *find_options*, které předá objekt *metafunc* a seznam jmen markerů, jejichž hodnotu chce uživatel získat.

Při selhání procesu získání hodnoty markeru způsobenou nevalidní hodnotou, typem, či použitím neznámého argumentu markeru, je vyvolána odpovídající výjimka, která uživatele informuje o lokaci a příčině vzniklé chyby, aby mohla být co nejrychleji nalezena a odstraněna.

6.6 Fixtury

Framework implementuje fixtury, které mohou být použity v testovacích případech. Tyto fixtury tvoří jednotné rozhraní pro testy a jsou navrženy tak, aby v případě změny jejich implementace, například změny referenčního modelu, byla nutná úprava frameworku minimální.

`def test_dir(request)`

Fixtura vrátí adresář zdrojového kódu aktuálního testovacího případu.

`def work_dir(request)`

Pracovní adresář testovacího případu. Před zahájením testovacího případu je vytvořen nový (nebo případně vyčištěn) pracovní adresář, kde dochází k vytváření dočasných souborů vzniklých během testovacího procesu, například přeložené zdrojové kódy assemblerovských testů.

`def compiler(work_dir)`

Fixtura vytvoří instanci překladače a tu vrátí. Překladač je vyhledáván v cestě, která je specifikována proměnnou prostředí *RISCV*. Ta obsahuje cestu k sadě nástrojů RISC–V. Dále je překladači nastaven pracovní adresář na pracovní adresář testu, aby nemusela být tato cesta explicitně nastavována v každém testovacím případu. Tím pádem je práce s překladačem zjednodušena.

`def model_path(request)`

Fixtura vrátí cestu k modelu procesoru, který má být otestován.

`def reference_model(work_dir)`

Fixtura vytvoří instanci referenčního modelu, kterým je v současnosti simulátor Spike.

`def toolchain_path(request)`

Vrátí cestu k sadě nástrojů, která bude využita pro překlad zdrojových souborů testů. Přeložené soubory budou simulovány na modelu procesoru, který je testován. Pokud uživatel tuto cestu nezadal, pak je použita sada nástrojů RISC–V extrahovaná z proměnné prostředí *RISCV*.

`def environment_path(request)`

Fixtura vrátí cestu k prostředí obsahující hlavičkové soubory pro překlad zdrojových souborů. Toto prostředí je součástí zásuvného modulu pro framework.

`def environment(request)`

Fixtura vrátí instanci třídy *Environment*, která uchovává cesty k hlavičkovým souborům, které jsou použity pro překlad. Tyto hlavičkové soubory jsou použity pro referenční překladač a referenční model.

6.7 Implementace testů

Zdrojové soubory testů nejsou implementovány v jazyce C, nýbrž v assembleru. Důvod je ten, že při použití jazyka C by navíc musel být testován překladač. Ten totiž může kód optimalizovat různými způsoby, a tak by mohl výsledný spustitelný soubor obsahovat i nepovolené instrukce – instrukce, které nepatří do testovaného standardního rozšíření. Při implementaci testů v assembleru tento problém odpadá, neboť překladač, resp. assembler má jasně definované instrukce, které má přeložit.

Zdrojové soubory jsou rozděleny do adresářů podle rozšíření, které testují. V současné době je implementována sada testů, obsahující testy pro základní instrukční sadu. Tato sada byla vytvořena pracovní skupinou v organizaci RISC–V Foundation a je použita pro experimentální ověření funkčnosti pro jeden instrukční simulátor a jednu hardwarovou implementaci RISC–V procesoru. Další typy testů, ověřující například nezarovnaný přístup do paměti, vyvolání výjimek, různé módy procesoru a další vlastností budou přibývat jakmile budou známy veškeré požadavky na tyto testy vyplývající ze standardu RISC–V. Na podobu testů má vliv pracovní skupina v organizaci RISC–V Foundation.

6.7.1 Testovací případ ve frameworku

Framework implementuje logiku, která vybírá testy, jenž mají být pro testovaný model procesoru spuštěny. Jako vstup framework očekává cestu ke zdrojovým souborům implementovaných v assembleru. Každý testovací případ implementovaný ve frameworku (tedy v jazyce Python) je pak dle použitých markerů, jejichž popisu je věnována sekce 6.5, spuštěn pouze s těmi zdrojovými soubory, pro které je daný testovací případ implementován. Selekci lze provádět zadáním relativní cesty od adresáře obsahující veškeré testy a použitím regulárních výrazů, kterých může být definováno více.

Filtrace testů je možná také pomocí vlastností testovaného modelu. Podpora filtrace podle vlastností modelu je nezbytná proto, aby mohlo být implementováno několik testů, které testují jedno rozšíření, ale s různými daty. Jeden test může testovat operaci dělení s validními argumenty a druhý například dělení nulou, kde je očekáváno vyvolání výjimky.

Každý testovací případ provede kompilaci zdrojového souboru jak pro referenční, tak testovaný procesor. Uživatel může pro překlad použít svou vlastní sadu nástrojů, a proto má možnost implementovat překlad speciálně přizpůsobeny jeho překladači. Obálka pro překladač, která je ve frameworku dostupná je kompatibilní s překladači odvozených od GCC, protože GCC je využíváno sadou nástrojů RISC–V, která slouží pro překlad souborů pro referenční model. Pokud překlad nebyl úspěšný, pak je testovací případ ukončen s chybou. V opačném případě jsou přeložené soubory spuštěny na referenčním instrukčním simulátoru Spike a testovaném procesoru. Opět, stejně jako v případu překladače, je na uživateli ponechána implementace metody pro simulaci aplikace na testovaném procesoru. Framework poskytuje pouze obecnou obálku pro spouštění podprocesu, avšak nezná konkrétní implementaci procesoru. Po dokončení simulace je z metody, která simulaci provádí, očekávána návratová hodnota, která obsahuje signaturu, jež je porovnána se signaturou referenčního modelu. Tato signatura obsahuje otisk z paměti, na které se nacházejí hodnoty architekturálních registrů. Do budoucnosti je předpokládáno, že návratová hodnota ze simulace bude obsahovat více než pouhou signaturu s hodnotami registrů, ale například i vyvolanou výjimku nebo jiné atributy. Proto se ze simulační metody nevrací pouze signatura, ale objekt, jehož atributem je signatura. Tím pádem bude možné objekt v budoucnosti rozšířit o další výsledky ze simulace. Při shodě signatur referenčního i tetovaného modelu je test označen

za procházející a přechází se k dalšímu testování. Ukázka testu je zobrazena na příkladu č. 6.3. Testovací případ využívá výše popsané možnosti pro selekci testů.

```
@pytest.mark.find_files(path=os.path.join('rv32i', 'ISA','src'),
                        pattern='\.S$')
@pytest.mark.architecture(isa=ISAS.RV32I)
def test_rv32i_isa(file, reference_model, compiler, environment,
                   user_model, user_compiler, user_environment,
                   work_dir):

    exe_ref = os.path.join(work_dir, os.path.basename(file)
                           +'.ref.xexe')
    exe_user = os.path.join(work_dir, os.path.basename(file)
                            +'.test.xexe')

    # Generate executable for reference model
    compiler.run(file, cflags=['-march=rv32i', '-mabi=ilp32',
                               '-o', exe_ref],
                       includes=[environment.path],
                       linker_script=environment.linker_script)
    # Generate executable for user model
    user_compiler.run(file, includes=[user_environment.headers],
                      cflags=['-o', exe_user])

    ref = reference_model.run(exe_ref, 'rv32i', work_dir)
    actual = user_model.run(exe_user, 'rv32i', work_dir)

    assert ref.signature == actual.signature,"Signature mismatch"
```

Příklad 6.3: Ukázka testovacího případu

6.7.2 Zdrojové soubory

Zdrojové soubory testu jsou implementovány v jazyce assembler. Tento jazyk byl upřednostněn před jazykem C, neboť při použití jazyka C by musely byt implementovány testy pro daný překladač, které by ověřovaly, ze spustitelný soubor obsahuje pouze povolené instrukce, tedy instrukce, které jsou dostupné pro standardní rozšíření, jež daný test ověřuje. Testování překladačů není v současnosti cílem testování kompatibility procesoru RISC–V. Jazyk C by přinášel také další problém, a to ten, ze ne každý návrhář procesoru má k dispozici překladač jazyka C. Tím pádem by byly tyto modely z hlediska frameworku netestovatelné. Minimálním obsahem sady nástrojů jsou tedy nástroje assembler a linker, které jsou běžně používány pro programování procesorů a mikrokontrolérů.

Testy obsahují pouze instrukce, které musí být implementovány, aby test mohl procházet. Pro kompatibilitu modelu s RISC–V standardem musí být vždy implementovány všechny instrukce základní instrukční sady, která obsahuje například instrukce pro práci s pamětí, základní aritmetické a logické operace, skokové instrukce a další. Bez těchto instrukcí by nebylo možné provádět kroky nezbytné pro inicializaci a provedení testu.

Testy ve svých zdrojových kódech používají makra, která jsou nahrazena při překladu. Použití maker je nezbytné z toho důvodu, že předem není známa implementace testovaného

procesoru. Různé implementace procesorů mají jinak uspořádaný adresový prostor, různě implementované fáze prolog a epilog, což jsou fáze pro inicializaci a ukončení programu. Prolog typicky obsahuje skok na resetovací vektor, který uvede procesor do stavu, aby byl schopný vykonávat program. Adresa resetovacího vektoru je závislá na implementaci procesoru a zná ji návrhář procesoru. Fáze epilog naopak ukončuje činnost procesoru. Bez této fáze by procesor mohl pokračovat v načítání dalších dat z paměti i po provedení všech instrukcí v programu – procesor by se snažil načítat data z paměti a ta interpretovat jako další instrukce. I tato fáze je implementačně závislá, a tudíž ji není možné implementovat přímo v testu, který má procházet na různých procesorech.

Makra jsou určena k implementaci uživatelem frameworku. Hlavičkový soubor s makry musí být ve složce *environment*, která je součástí zásuvného modulu pro framework, který je popsán v sekci 6.3.

Framework v současné chvíli vyžaduje přítomné tři hlavičkové soubory:

- *encoding.h*
- *riscv_test.h*
- *compliance_test.h*

Soubor *encoding.h* obsahuje definici základních konstant a masek definovaných standardem. Masky slouží například k extrakci bitů stavových registrů. Veškeré konstanty a masky definované v tomto souboru, jsou definované v popisu standardu RISC–V, a tudíž jsou společné pro všechny implementace procesorů RISC–V. Dále definuje všechny stavové registry a standardně podporované výjimky. Tento soubor je v aktuálně dostupný ve veřejném repositáři RISC–V, ale v budoucnosti je plánováno, že bude součástí frameworku, neboť je nezbytný pro jeho funkcionalitu a není ani závislý na implementaci procesoru. Na příkladu 6.4 je ukázána část souboru *encoding.h* zobrazující definici konstant pro režimy procesoru. V pořadí jde o definici hodnot pro uživatelský režim (*PRV_U*), režim supervizora (*PRV_S*), rezervovanou hodnotu (*PRV_H*) a strojový režim (*PRV_M*).

```
#define PRV_U 0
#define PRV_S 1
#define PRV_H 2
#define PRV_M 3
```

Příklad 6.4: Ukázka encoding.h

Soubor *riscv_test.h* obsahuje definici základních maker, která jsou využívána ve všech testovacích případech. Jde o makra sloužící pro inicializaci kódu, ukončení činnosti procesoru, inicializaci datové části, označení části paměti, kde se nachází signatura. Signatura je blok paměti, který je důležitý pro testování kompatibility se standardem RISC–V. Právě tento blok paměti je po provedení všech instrukcí v testu extrahován a porovnán se signaturou referenčního modelu. Signatura obsahuje hodnoty jednotlivých architekturálních registrů, jejichž obsah je modifikován při provádění instrukcí. Některé důležitá makra jsou například:

- *RVTEST_CODE_BEGIN* – makro značící začátek programu. Obsahuje inicializaci procesoru, která bývá typicky prováděna skokem na resetovací vektor.
- *RVTEST_CODE_END* – makro značící konec programu. V některých případech může zůstat prázdné, avšak mohou existovat procesory, které potřebují pro své ukončení provést jednu či více instrukcí.

- *RVTEST_DATA_BEGIN* – makro značící začátek bloku paměti obsahující výslednou signaturu. Může obsahovat symboly, které jsou v abstraktním modelu využity pro získání signatury po skončení simulace programu.
- *RVTEST_DATA_END* – makro značící konec bloku paměti obsahující výslednou signaturu.

Podobně jako u souboru *encoding.h*, je tento soubor součástí repositáře RISC–V a předpokládá se jeho přesun do frameworku, avšak tuto operaci lze provést až po schválení v pracovní skupině organizace RISC–V Foundation.

Soubor *compliance_test.h* obsahuje definici maker, která přetěžují makra definovaná ve výše popsaném souboru *riscv_test.h*. V tomto souboru by měl uživatel definovat makra specifická pro své procesorové jádro.

Uživatel má možnost přidání vlastního skriptu pro linker v případě, že je jeho použití nutné. Skript pro linker obsahuje informace pro nástroj linker, který vytváří spustitelný soubor. Tento skript popisuje způsob, jakým by měly být namapovány sekce v konečném spustitelném souboru. Určuje například vstupní bod programu, od kterého má být zahájeno provádění programu (standardně jde o symbol *_start*).

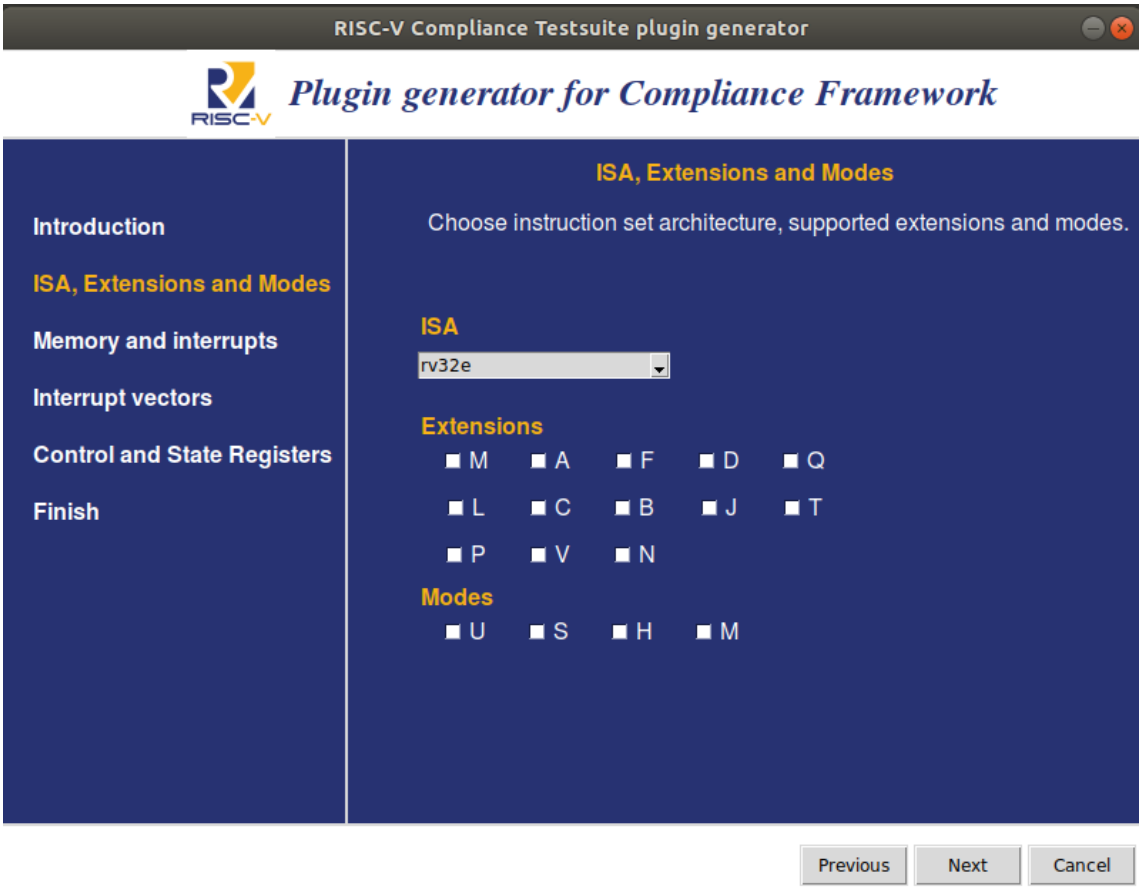
6.8 Uživatelské rozhraní

Sekundárním cílem je vytvořit uživatelské rozhraní, které uživateli zjednoduší práci s frameworkem. Nejnáročnější částí při používání tohoto frameworku je prvotní nastavení, tedy vytvoření kompatibilního zásuvného modulu obsahujícího implementaci abstraktního modelu. Proto je pro generátor zásuvných modulů vytvořeno grafické i textové rozhraní.

Grafické uživatelské rozhraní je implementováno formou průvodce. Vlastnosti abstraktního modelu jsou tedy rozděleny do několika logických sekcí a uživatel vybírá z nabízených možností ty, které odpovídají jeho modelu procesoru. Tyto sekce jsou znázorněny na obr. č. 6.6. Pro implementaci byla zvolena grafická knihovna *Tkinter*, která je jednoduchá na použití a zároveň je součástí standardní distribuce jazyka Python. Tím pádem je knihovna dostupná na všech systémech, pro které existuje interpret jazyka Python.

Některá pole vyžadují zadání hodnoty uživatelem. V tomto případě je při zadávání hodnoty prováděna kontrola validity těchto hodnot. Pokud uživatel zadá nevalidní hodnotu, pak je prostřednictvím grafického uživatelského rozhraní informován o této skutečnosti. Zadávání je opakováno do té doby, dokud nejsou všechny hodnoty validní. V opačném případě nelze generování zásuvného modulu dokončit, neboť by tento modul neobsahoval validní model. Sekce v uživatelském rozhraní jsou následující:

- *Introduction* – Úvodní sekce popisující generátor zásuvných modulů.
- *ISA and Extensions* – Zvolení architektury instrukční sady dle standardu RISC–V a podporovaných standardních rozšíření.
- *Memory and Interrupts* – Nastavení velikosti paměti, počáteční adresu programu a počáteční adresu pro data. Zároveň je zde zvolena možnost, zda procesor podporuje přerušení a přístupy do nezarovnané paměti.
- *Interrupt Vectors* – Zvolení výčtu podporovaných přerušení.
- *Control and State Registers* – Zvolení výčtu implementovaných kontrolních a stavových registrů.



Obrázek 6.6: Grafické uživatelské rozhraní

- *Finish* – Vybrání adresáře, kam má být zásuvný modul vygenerován a samotné započetí generování zásuvného modulu.

Druhým implementovaným rozhraním je textové uživatelské rozhraní. To slouží pro uživatele preferující tento typ rozhraní a také má výhodu, že nepotřebuje dodatečnou instalaci grafické knihovny *Tkinter*, která je sice standardní knihovnou, ale její instalace je volitelná; nelze tudíž předpokládat, že bude v každém případě nainstalována na uživatelově počítači. Požadované parametry abstraktního modelu jsou shodné se parametry v grafickém uživatelském rozhraní. Ukázka podoby textového uživatelského rozhraní je zobrazena na obr. č. 6.7.

```
Set ISA architecure (required): rv32i
Select supported standard extensions (optional): a, m, f
Select supported modes (required): M
Set memory (size, program_start, data_start) (required): 0x4000, 0x1000, 0x2000
Misaligned memory access support [y/N]: y
Interrupt support [y/N]: y
Select supported causes (optional): misaligned fetch, misaligned load
Select supported csrs (optional):
```

Obrázek 6.7: Textové uživatelské rozhraní.

6.9 Parametrizace frameworku

Frameworku je možné předat argumenty, které slouží ke správné inicializaci. Některé parametry jsou povinné, ale cílem je mít takových argumentů, co nejméně, aby nebyl uživatel nucen používat přílišné množství argumentů. V textu práce jsou popsány pouze některé důležité argumenty, neboť výčet všech argumentů není pro výklad podstatný. Mezi povinné argumenty patří:

- –plugin – cesta k zásuvnému modulu obsahující abstraktní model. Tento modul mohl být předem vygenerován generátorem zásuvných modulů, který je popsán v sekci 6.3.1.
- –model – cesta k modelu, který implementuje procesor RISC–V. Tento model je použit k simulaci testů.
- –environment – cesta k prostředí obsahující hlavičkové soubory pro referenční model. V současnosti jsou tyto hlavičky součástí veřejného repositáře RISC–V, a proto je nelze načítat automatickým způsobem. Do budoucna se ale počítá s přesunem těchto souborů do frameworku a odstranění tohoto argumentu.

Framework poskytuje i další argumenty, které slouží pro usnadnění práce s frameworkem nebo přizpůsobení prostředí. Do této skupiny se řadí argumenty pro spuštění generátoru zásuvných modulů (v grafickém nebo textovém režimu), možnost vygenerování referenční sady nástrojů a referenčního simulátoru Spike, změna implicitního pracovního adresáře a další.

6.10 Hlášení

Hlášení je vygenerováno vždy po skončení činnosti frameworku. Cílem je informovat uživatele o výsledcích testů. Během procesu testování jsou na obrazovku průběžně vypisovány výsledky jednotlivých testů, avšak tyto výpisy slouží pouze pro rychlou orientaci a neposkytují detailní informace. Ty jsou dostupné až ve vygenerovaných hlášeních ve formátu HTML a XML. Obě hlášení obsahují identická data, avšak formát HTML je vhodnější pro čtení člověkem a formát XML je vhodnější pro strojové zpracování. V případě selhání testu je navíc obsah pracovního adresáře, ve kterém se mohou nacházet dočasné soubory, uchován pro účely ladění. Tento obsah se nachází ve složce *failed*, který je automaticky vytvořen v pracovním adresáři frameworku.

Testy jsou v HTML hlášení tříděny do skupin podle jejich výsledku. V prvním sloupci je uveden výsledek testu; test mohl skončit jedním z následujících způsobů:

- *Passed* – test prošel dle očekávání; signatura referenčního modelu a testovaného modelu je shodná.
- *Skipped* – test nebyl na testovaném modelu spuštěn.
- *Failed* – test byl označen za selhávající; signatura referenčního modelu a testovaného modelu je různá nebo se nepodařilo přeložit zdrojový soubor testu.
- *Error* – během provádění testu došlo k nečekané chybě.

- *Expected failure (xfail)* – během vykonávání testu bylo očekáváno vyvolání výjimky, které nastalo.
- *Unexpected pass* – během vykonávání testu bylo očekáváno vyvolání výjimky, které ale nenastalo.

Ve druhém sloupci se nachází cesta k testu, kterému daný výsledek (řádek) odpovídá včetně jeho argumentů. Cesta testu společně s argumenty tvoří jedinečný identifikátor testovacího případu. Ve třetím sloupci je zobrazena doba vykonávání testu v sekundách. Tato doba v sobě zahrnuje veškeré provádění testovacího případu, tj. kompilace zdrojového souboru pro referenční a testovaný model a zároveň simulaci aplikace na obou modelech.

Uživatel má možnost si filtrovat testy podle typu výsledku, které jsou vyjmenovány výše. U přeskočených testů je uveden i důvod, proč test nebyl spuštěn – nejčastějším důvodem bude to, že model nepodporoval nějakou vlastnost, kterou test požadoval. U selhávajících testů lze po zobrazení detailů spatřit příčinu selhání testu. Hlášení každého testu, který selhal obsahuje zachycený standardní výstup a standardní chybový výstup, pokud nějaký zachycen byl. Ke každému testovacímu případu je možné během testovacího procesu přidávat metadata, která jsou v hlášení zobrazena také. Standardní výstup, standardní chybový výstup a metadata může uživatel využít k ladění modelu. Ukázky hlášení jsou zobrazeny na obrázcích č. 6.8 a 6.9.

Summary

52 tests ran in 6.10 seconds.

(Un)check the boxes to filter the results.

☑ 52 passed, ☑ 0 skipped, ☑ 0 failed, ☑ 0 errors, ☑ 0 expected failures, ☑ 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-CSRRW-01.S]	0.10
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-XORI-01.S]	0.10
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-SRAI-01.S]	0.10
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-ECALL-01.S]	0.08
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-BGE-01.S]	0.13
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-SLL-01.S]	0.10
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-XOR-01.S]	0.10
Passed *(show details)*	git/rvtest/compliance_tests/rv32i/ISA/test_isa.py::test_rv32i_isa[I-SRL-01.S]	0.10

Obrázek 6.8: Hlášení ve formátu HTML.

Obrázek 6.9: Hlášení ve formátu XML.

Kapitola 7

Experimentování

Cílem práce je framework experimentálně ověřit na jednom instrukčním simulátoru a na jedné hardwarové implementaci. Tyto modely, včetně sady nástrojů, které jsou potřebné pro překlad zdrojových souborů testů byly poskytnuty společností Codasip, která se zabývá návrhem procesorových architektur a automatizací generování sady nástrojů, které jsou s těmito procesory kompatibilní. Sada nástrojů je založena na technologiích GCC a LLVM[1]. Jeden z procesorů, které Codasip vyvíjí, je Codix Berkelium, který je implementován dle standardu RISC–V a byl použit pro ověření správně funkčnosti frameworku. Jsou použity 2 implementace tohoto modelu. První je instrukční simulátor, který pouze simuluje jednotlivé instrukce programu a zanedbává přitom hodinové takty. Druhou implementací je model, který časovou složku, resp. hodinové takty respektuje. Anglicky se tyto typy modelů označují jako *cycle accurate*. Tuto implementaci lze označit za hardwarovou, neboť simuluje chování procesoru tak, jak by se dělo ve skutečném hardwaru, tedy kromě hazardů, výkyvů napájecího napětí a dalších problémů, které vznikají na polovodičových čipech. Pro účely experimentálního otestování frameworku je ale tato implementace dostačující.

Aby bylo možné modely pomocí frameworku otestovat, je nutné vygenerovat zásuvný modul, který je popsán v sekci 6.3. Instrukční simulátor i hardwarová implementace procesoru Codix Berkelium mají stejné rozhraní, proto stačí vygenerovat pouze jeden zásuvný modul a ten použít jak pro instrukční simulátor, tak hardwarový model. K vygenerování zásuvného modulu bylo použito textové uživatelské rozhraní a vlastnosti abstraktního modelu jsou uvedeny na obr. č. 7.1. Část vygenerovaného modulu, která provádí inicializaci abstraktního modelu, lze spatřit na obr. č. 7.2.

```
jenkins@jenkins-PC:~/git/rvtest$ python3 rvtest.py --generator
Set ISA architecure (required): rv32i
Select supported standard extensions (optional):
Select supported modes (required): M
Set memory (size, program_start, data_start) (required): 0x100000,0x1000,0x2000
Misaligned memory access support [y/N]: N
Interrupt support [y/N]: y
Select supported causes (optional): misaligned fetch,fetch access,illegal instruction
Select supported csrs (optional): mtvec,mtval,mstatus,mepc,mie,mcause,mscratch
Select destination path (required): /home/jenkins/git/rvtest/rvtest_plugin
info: Generating plugin
info: Creating output directory /home/jenkins/git/rvtest/rvtest_plugin
jenkins@jenkins-PC:~/git/rvtest$
```

Obrázek 7.1: Generování zásuvného modulu

[1] Low Level Virtual Machine

```
model = RISCVComplianceModel(model_path)

# Specify ISA
model.add_property(PlatformProperties.ISA, ISAS.RV32I)
# Specify supported modes
model.add_property(PlatformProperties.MODE, Modes.MACHINE)
# Specify RISCV extensions

# Set memory range. Tuple of (memory_size, program_start, data_start)
model.add_property(PlatformProperties.MEMORY_RANGE, [1048576, 4096, 8192])
# Does misaligned memory access support
model.add_property(PlatformProperties.MEMORY_MISALIGNED, False)
# Set interrupt support
model.add_property(PlatformProperties.INTERRUPT_SUPPORT, True)
# Set supported exception causes
model.add_property(PlatformProperties.CAUSE, Causes.FETCH_ACCESS)
model.add_property(PlatformProperties.CAUSE, Causes.MISALIGNED_FETCH)
model.add_property(PlatformProperties.CAUSE, Causes.ILLEGAL_INSTRUCTION)
# Set implemented control and status registers
model.add_property(PlatformProperties.CSR, CSRS.MSTATUS)
model.add_property(PlatformProperties.CSR, CSRS.MIE)
model.add_property(PlatformProperties.CSR, CSRS.MEPC)
model.add_property(PlatformProperties.CSR, CSRS.MTVEC)
model.add_property(PlatformProperties.CSR, CSRS.MSCRATCH)
model.add_property(PlatformProperties.CSR, CSRS.MCAUSE)
model.add_property(PlatformProperties.CSR, CSRS.MTVAL)
```

Obrázek 7.2: Inicializace abstraktního modelu

Ve vygenerovaném zásuvném modulu je potřeba upravit metodu abstraktního modelu, která slouží pro spuštění simulace a extrakci signatury. V metodě pro spuštění simulace je nutné změnit argumenty pro simulátor tak, aby po skončení simulace došlo k výpisu signatury na standardní výstup. Standardní výstup je během simulace zachytáván, a proto je jeho získání po dokončení simulace snadné. Referenční simulátor Spike vrací signaturu jako řetězec znaků v kódování UTF-8[2], avšak standardní výstup ze simulace je zachytáván jako posloupnost binárních symbolů, proto je potřeba tuto posloupnost převést na řetězec v požadovaném kódování. Úpravy kódu abstraktního modelu jsou ukázány na příkladu 7.1. Modely jsou kompatibilní s debbugerem GDB[3], čili poskytují podobné rozhraní na příkazové řádce. Pro spuštění simulace bez interaktivního režimu je přidán argument *–r* a pro vypsání signatury na standardní výstup slouží argument *–info 5*, který není podporován debuggery GDB, ale je součástí implementace modelu Codix Berkelium.

```
def get_signature(self, process_result):
    return process_result.stdout.decode('utf-8')

def run(self, file, extension):
    args = ['-r', file, '--info', '5']
```

Příklad 7.1: Úprava abstraktního modelu

Sada nástrojů, která je použita pro překlad zdrojových souborů pro model Codix Berkelium je kompatibilní s překladačem GCC, a proto není potřeba žádná úprava vygenerova-

[2]Unicode Transformation Format
[3]GNU Debugger

ného kódu obálky pro překladač. Avšak, aby byl překlad zdrojových souborů testů úspěšný, musela být upravena makra v hlavičkových souborech (viz. podkapitola 6.7.2), která provádí inicializaci procesoru, neboť model má jiný průběh inicializace, než referenční model Spike.

Po spuštění frameworku s instrukčním simulátorem bylo zahájeno testování základní instrukční sady RV32I, pro kterou v současnosti testy existují. Tato sada testů obsahuje celkem 52 zdrojových souborů, kde každý test testuje jinou instrukci ze základní sady instrukcí. Testovací proces trval 6,14 sekund pro instrukční simulátor a 6.26 sekund pro hardwarovou implementaci. Všechny testy byly označeny jako procházející, tedy v každém z testů byla signatura instrukčního modelu shodná s referenčním simulátorem Spike. Výsledky testovacího procesu lze spatřit na obrázcích č. 7.3 a 7.4.

```
========================= test session starts ==========================
platform linux -- Python 3.6.3, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
ISA: rv32i
Toolchain: /home/jenkins/test/tools/bin/
RISC-V model path: /home/jenkins/codasip/ia/sdk/bin/codix_berkelium-
ia-isimulator
Reference environment: /home/jenkins/riscv-env/p
rootdir: /home/jenkins, inifile:
plugins: html-1.17.0, metadata-1.7.0
collected 52 items

git/rvtest/compliance_tests/rv32i/ISA/test_isa.py ..................
................................

- generated xml file: /home/jenkins/rvtest_work/report/report.xml --
 generated html file: /home/jenkins/rvtest_work/report/report.html -
Codix Berkelium is compliant with RISC-V standard
==================== 52 passed in 6.14 seconds =====================
```

Obrázek 7.3: Výsledek testování kompatibility IA modelu

```
========================= test session starts ==========================
platform linux -- Python 3.6.3, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
ISA: rv32i
Toolchain: /home/jenkins/test/tools/bin/
RISC-V model path: /home/jenkins/codasip/ca/sdk/bin/codix_berkelium-
ca-isimulator
Reference environment: /home/jenkins/riscv-env/p
rootdir: /home/jenkins, inifile:
plugins: html-1.17.0, metadata-1.7.0
collected 52 items

git/rvtest/compliance_tests/rv32i/ISA/test_isa.py ..................
................................

- generated xml file: /home/jenkins/rvtest_work/report/report.xml --
 generated html file: /home/jenkins/rvtest_work/report/report.html -
Codix Berkelium is compliant with RISC-V standard
==================== 52 passed in 6.26 seconds =====================
```

Obrázek 7.4: Výsledek testování kompatibility CA modelu

Kapitola 8

Závěr

Cílem práce bylo navrhnout a implementovat systém pro spouštění testů kompatibility, který ověří, že implementovaný model procesoru odpovídá standardu RISC–V. Pro implementaci tohoto systému byl zvolen programovací jazyk Python 3.6 s testovacím frameworkem Pytest, který umožňuje snadnou implementaci vlastního zásuvného modulu do tohoto frameworku. Zásuvný modul pro Pytest může ovlivňovat všechny fáze testovacího procesu používáním předefinovaných i vlastních hooků. Tímto způsobem je implementován i framework, jež byl cílem práce.

Framework umožňuje uživateli vytvořit abstraktní model, který popisuje model procesoru, jež bude použit pro ověření kompatibility se standardem RISC–V. Tento abstraktní model je implementován pomocí zásuvného modulu pro framework a ten jej využívá pro selekci podmnožiny testů, které testují různé požadavky definované RISC–V standardem. Úkolem této práce bylo uživateli umožnit vytvoření tohoto abstraktního modelu co možná nejjednodušším způsobem. Proto je implementován generátor zásuvného modulu, který obsahuje definici abstraktního modelu a náležité moduly a soubory, mezi které patří implementace obálky pro překladač. Tím pádem může uživatel snadno použít svou vlastní sadu nástrojů. Pro překlad zdrojových souborů je využita sada nástrojů poskytovaná organizací RISC–V Foundation. Ta je dostupná ve veřejném repositáři. Framework navíc umožňuje automatické vytvoření této sady nástrojů ze zdrojových souborů. To usnadňuje uživateli prvotní nastavení frameworku. Pro generátor zásuvných modulů je vytvořeno grafické uživatelské rozhraní, které je implementováno formou průvodce. Proces generování je tedy rozdělen na několik kroků, které jsou logicky odděleny. Nejprve je uživatelem zvolena architektura instrukční sady a podporovaných standardních rozšíření. Poté následuje definice ostatních potřebných vlastností modelu. Díky grafickému uživatelskému rozhraní není uživatel nucen studovat aplikační rozhraní frameworku.

Po dokončení činnosti frameworku, je uživateli poskytnuta informace, zda je implementovaný model procesoru kompatibilní s RISC–V standardem. Nutno poznamenat, že ověřování kompatibility je prováděno vůči určité verzi standardu. Jeho podoba se totiž může a velmi pravděpodobně bude měnit, je to stále dynamicky se vyvíjející standard. Dalším výstupem jsou hlášení ve formátu HTML a XML. Formát HTML je vytvořen, protože je snadno čitelný pro člověka – lze ho zobrazit jako stránku ve webovém prohlížeči. Toto hlášení obsahuje i kaskádové styly pro maximální uživatelskou přívětivost. V hlášení jsou u každého testu, který selhal, ke spatření signatury uživatelova a referenčního modelu. Uživatel tedy má možnost zjistit, jaká byla očekávaná hodnota. Tuto informaci může využít pro ladění svého modelu. Druhým hlášením je formát XML, který slouží pro strojovou

analýzu. To může využít uživatel k dlouhodobému uchovávání výsledků, které je vhodné pro regresní testování.

Framework byl navržen tak, aby mohlo snadno docházet k budoucím rozšířením. Mezi tato rozšíření patří hlubší analýza příčiny, proč některý z testů spadl. Kdyby framework tuto vlastnost podporoval, bylo by pro uživatele snazší upravit jeho model tak, aby daný test procházel, a tím pádem jej více přiblížil požadavkům standardu RISC–V. Dalším možným rozšířením je podpora testování modelů RISC–V procesoru na FPGA[1]. FPGA desky by mohly být připojeny k počítači pomocí JTAG[2] rozhraní a framework by testoval kompatibilitu čipu na FPGA se standardem RISC–V.

8.1 Vývoj v pracovní skupině RISC–V Foundation

Tato podkapitola závěru popisuje průběh vývoje frameworku v pracovní skupině, která se věnuje testování kompatibility a pro kterou byl tento framework implementován. Autor práce je členem této pracovní skupiny, aby mohl sledovat aktuální vývoj a k tomuto vývoji přispívat. Skupina pravidelně pořádá sezení, kde probíhá diskuze nad aktuálním stavem v ostatních skupinách a kde jsou diskutovány požadavky na framework, který je výsledkem této práce. Kvůli faktu, že RISC–V Foundation je mezinárodní organizace složená ze spousty přispívajících členů, je proces schvalování, a tím pádem i celkový vývoj, pomalý. V průběhu vypracovávání práce (konec února 2018) došlo ke změně vedoucího skupiny, což mělo za následek pozastavení vývoje uvnitř skupiny. Vývoj, resp. diskuze, byly obnoveny v průběhu března letošního roku. Dosavadní vývoj byl z velké části zahozen a návrh podoby testů a frameworku byl restartován. V současné době existuje pouze velmi hrubý nástin podoby frameworku. Vzhledem k časovým požadavkům na termín odevzdání diplomové práce, autor navrhl framework, který plně respektuje veškeré součásti dosavadního návrhu pracovní skupiny, avšak většina implementační části musela být vytvořena bez existence specifikace požadavků ze strany pracovní skupiny. Autorův úplný návrh bude představen pracovní skupině v blízkém termínu, včetně zdůvodnění všech implementačních detailů. Následně bude probíhat diskuze, zda je autorův návrh vhodný, a případně budou diskutovány nezbytné změny, které budou potřebné pro splnění očekávání pracovní skupiny.

[1] Field-Programmable Gate Array – programovatelné hradlové pole

[2] Joint Test Action Group – standard popisující architekturu pro testování plošných spojů.

Literatura

[1] *Automated vs. Manual Testing: The Pros and Cons of Each* [online]. Base36, [cit. 2017-12-19].
URL http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/

[2] CS Division, EECS Department, University of California, Berkeley: *The RISC-V Instruction Set Manual Volume II: Privileged Architecture [online]*. may 2017, [cit. 2017-03-18].
URL https://riscv.org/specifications/privileged-isa/

[3] CS Division, EECS Department, University of California, Berkeley: *The RISC-V Instruction Set Manual Volume I: User-Level ISA [online]*. may 2017, [cit. 2017-12-09].
URL https://riscv.org/specifications/

[4] holger krekel and pytest-dev team: *Writing plugins [online]*. 2015, [cit. 2017-03-18].
URL https://docs.pytest.org/en/latest/writing_plugins.html

[5] *Automated Testing using BDT (Behavior Driven Testing)* [online]. únor 2013, [cit. 2018-01-15].
URL https://www.infostretch.com/blog/automated-testing-using-bdt-behavior-driven-testing/

[6] *Test Automation Frameworks* [online]. Smartbear, [cit. 2018-01-15].
URL https://smartbear.com/learn/automated-testing/test-automation-frameworks/

[7] *Firmware Updates and Initial Performance Data for Data Center Systems* [online]. Intel, leden 2018, [cit. 2018-01-17].
URL https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/

[8] *Conformance Testing* [online]. National Institute of Standards and Technology, září 2010, [cit. 2018-01-15].
URL https://www.nist.gov/itl/ssd/information-systems-group/conformance-testing

[9] *Types of Non Functional Software Testing - TestingWhiz* [online]. TestingWhiz, únor 2012, [cit. 2017-12-17].
URL https://www.testing-whiz.com/blog/types-of-non-functional-software-tests

[10] *Agile vs. Waterfall in Hardware Product Development* [online]. GRAVES Eric, únor 2016, [cit. 2017-12-17].
URL https://www.playbookhq.co/blog/agile-vs.-waterfall-in-hardware-product-development

Příloha A

Obsah DVD

Přiložené DVD obsahuje následující adresáře:

- */doc* - obsahuje technickou zprávu ve formátu pdf,
- */tex* - obsahuje soubory se zdrojovými kódy technické zprávy včetně použitých obrázků,
- */src* - obsahuje soubory se zdrojovými kódy výsledné aplikace. Struktura adresáře je detailněji popsána v souboru readme.txt, který se nachází v této složce.