VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZOVANÉ TESTOVÁNÍ V FPGA
AUTOMATIZATION OF TESTING ON FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID VALECKÝ**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2020

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Valecký David**
Program: Informační technologie
Název: **Automatizované testováni v FPGA**
Automated Testing in FPGA
Kategorie: Analýza a testování softwaru

Zadání:

1. Seznamte se s možnostmi testování v FPGA, s jejich rozhraním a způsobem obsluhy.
2. Seznamte se s druhy testů a vyberte ty, které by bylo vhodné provádět v FPGA zařízeních.
3. Analyzujte výhody testování na FPGA zařízeních.
4. Navrhněte systém pro spouštění testů na platformě FPGA.
5. Implementujte navržený systém a otestujte jeho výkon.
6. Zhodnoťte dosažené výsledky a navrhněte možná rozšíření.

Literatura:

- Wang, F., Wang, D., Yang, H., Xie, X., & Fan, D. (2016). On-Chip Generating FPGA Test Configuration Bitstreams to Reduce Manufacturing Test Time. *Chinese Journal of Electronics*, *25*(1), 64-70.
- Deng, H., Wang, J., Tao, X., & Lai, J. A Standalone FPGA-test Platform with Bi-directional Ports Supported. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)* (pp. 1-3). IEEE.
- Okken, B. (2017). *Python Testing with Pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Dolíhal Luděk, Ing., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 22. října 2020

Abstrakt

Cílem této práce je analyzovat testování procesorů vyvíjených firmou Codasip a zjistit, které z testů je vhodné provádět s využitím technologie FPGA. Dále je cílem navrhnout a implementovat systém na vzdálené konfigurování zařízení FPGA připojených k centrálnímu serveru za účelem provádění testů. Systém byl naprogramován v jazyce Python s využitím architektury klient-server a frameworku Flask. Interakce serveru se zařízeními FPGA je zajištěna s pomocí sofwaru OpenOCD. Implementované řešení umožňuje uživateli zjistit stav připojených obvodů FPGA, nakonfigurovat tato zařízení a následně je využít k běhu testů. V rámci práce byly využity obvody FPGA řady Artix-7 firmy Xilinx, umístěné na vývojových deskách Digilent Nexys A7. Výsledné testování naprogramovaných čipů v FPGA reprezentující mikroprocesor je urychleno při použití FPGA zařízení. Jeho výsledky jsou v některých případech na hardwarové reprezentaci rychlejší než při jeho simulaci.

Abstract

The aim of this work is to analyze the testing of processors developed by Codasip and find out which of the tests should be performed using FPGA devices. Furthermore, the goal is to design and implement a system for remote operation of FPGA devices connected to a central server in order to perform tests. The system is programmed in Python using the client-server architecture and Flask framework. The interaction of the server with the FPGA devices is ensured with the help of OpenOCD. The implemented system allows a user to find out the status of connected FPGA circuits, configure these devices and then use them to run tests. The work uses FPGAs Artix-7 series made by Xilinx, placed on Digilent Nexys A7 development boards. The resulting testing of programmed chips in an FPGA representing a microprocessor is accelerated when using an FPGA device. Its results are faster on hardware representation than on its simulation in some cases.

Klíčová slova

testování, Python, FPGA, webová aplikace, mikroprocesor, procesor

Keywords

testing, Python, FPGA, web application, microprocessor, processor

Citace

VALECKÝ, David. *Automatizované testování v FPGA*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Automatizované testování v FPGA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytli Ing. Jan Matyáš, Ing. Luděk Dolíhal, Ph.D. a Ing. Jiří Hynek, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
David Valecký
11. května 2021

Poděkování

Tímto bych chtěl poděkovat panu inženýrovi Janu Matyášovi za obrovskou podporu a pomoc při řešení práce. Dále děkuji panu doktoru Jiřímu Hynkovi za pravidelné konzultace a vlídné rady, panu doktoru Luďku Dolíhalovi za pomoc se zprostředkováním náležitých prostředků pro vykonávání práce a panu profesoru Tomáši Hruškovi za vedení práce. Také děkuji své rodině za psychickou a morální podporu a své přítelkyni, která při mně neustále stála, i když šlo opravdu leckdy o absolutní vyhoření v pozdních nočních hodinách.

Obsah

1 Úvod **2**

2 Procesory a FPGA **3**
2.1 Procesory . . . 3
2.2 RISC-V . . . 6
2.3 Abstrakce návrhu procesoru . . . 7
2.4 RTL . . . 9
2.5 FPGA . . . 12
2.6 JTAG . . . 15

3 Testování procesorů **16**
3.1 Funkční verifikace . . . 16
3.2 Formální verifikace . . . 19

4 Analýza **20**
4.1 Aktuální stav . . . 20
4.2 Testy reprezentace RTL procesorů firmy Codasip . . . 20
4.3 Důvody pro implementaci systému . . . 24
4.4 Existující řešení . . . 24

5 Návrh testovacího systému pro farmu FPGA **27**
5.1 Architektura systému . . . 27
5.2 Způsob využití systému . . . 28
5.3 Aplikační rozhraní . . . 29
5.4 Databázové schéma . . . 30

6 Implementace **32**
6.1 Klient . . . 32
6.2 Server . . . 33

7 Testování systému **37**
7.1 Manuální testování . . . 37
7.2 Systém integrovaný v testovacím frameworku . . . 37
7.3 Budoucí vylepšení systému . . . 38

8 Závěr **40**

Literatura **41**

Kapitola 1

Úvod

Procesory jsou základní komponentou výpočetních systémů, ať už se jedná o osobní počítače, servery nebo vestavěné systémy (angl. *embedded systems*). Značnou část těchto procesorů používaných v oblasti vestavěných systémů v současnosti vyvíjí firma *ARM Ltd* – procesory se stejnojmennou architekturou ARM. Společnost procesory nevyrábí, ale prodává zpoplatněné licence poskytující jejich model obsahující architekturu procesoru a instrukční sadu. V oblasti osobních počítačů a serverů se uplatňují převážně procesory s instrukčními sadami z rodiny Intel x86.

V roce 2010 byla na Kalifornské univerzitě v Berkeley navržena a publikována architektura instrukční sady *RISC-V*. Specifikace RISC-V popisuje instrukční sadu, nikoli vnitřní architekturu procesorů. Narozdíl od instrukčních sad ARM a Intel x68 je specifikace RISC-V tzv. otevřený standard. Umožňuje tedy implemetaci jak otevřených (angl. *open-source*) implementací procesorů, tak i komerčních (proprietárních) procesorů, a to bez nutnosti hradit licenční poplatky. Specifikaci RISC-V spravuje a zaštiťuje nezisková asociace jménem RISC-V International, jejímiž členy je už přes 700 technologických firem, například velké firmy Google, Western Digital nebo NVidia. Mezi členy je i firma Codasip vyvíjející procesory založené na RISC-V, která je zadavatelem této práce.

Při verifikaci návrhů procesorů se obvykle využívají metody softwarové simulace, neboť výroba prototypu procesoru (ve formě polovodičové součástky – integrovaného obvodu) je vysoce finančně nákladná a časově náročná. Některé testy procesoru je možné provést jiným způsobem – za pomoci obvodů *FPGA* (angl. zk. *Field-Programmable Gate Array*). Jedná se o programovatelné digitální obvody, které nemají z výroby pevně danou funkci, ale jejich chování může určit uživatel. Jedná se o programovatelný logický integrovaný obvod. Chování obvodu FPGA se definuje nahráním speciálního binárního souboru, který reprezentuje konkrétní digitální obvod, například navržený procesor.

Tato práce implementuje softwarový systém pro správu FPGA zařízení připojených k centrálnímu počítači, který umožní správu zdrojů a přidělování FPGA zařízení jednotlivým běžícím úkolům. Klade také za cíl identifikovat, které testy procesorů, používaných ve firmě Codasip, je možné a vhodné provádět na zařízeních FPGA. Pro tyto účely firma využívá FPGA čipy Artix-7 od firmy Xilinx, umístěné na vývojových deskách Nexys A7-100T od výrobce Digilent.

V první části práce jsou shrnuty základní pojmy z oblasti týkající se této práce, zejména se jedná o procesory, instrukční sadu RISC-V a technologie FPGA. Na to navazuje popis zařízení FPGA, následuje testování právě RISC-V procesorů na FPGA zařízeních a poté je uveden systém pro testování a jeho implementace. Posléze je provedeno testování systému pro ověření očekávaných vylepšení, které má systém přinést.

Kapitola 2

Procesory a FPGA

Kapitola seznamuje čtenáře s pojmy z oblasti procesorů a s úrovněmi abstrakce, které se při popisu procesorů využívají. Větší pozornost je zde věnovaná úrovni RTL (angl. zk. *Register-transfer level*). Jedná se o základní abstrakci dnes používanou pro definování elektronických systémů sloužící jako jejich vhodný model návrhu a ověřování. Následně je přiblížena architektura instrukční sady RISC-V. V závěru kapitoly jsou představeny obvody typu FPGA a je vysvětlena motivace jejich použití v procesu testování.

2.1 Procesory

Procesor je digitální obvod, který provádí operace nad daty uloženými v paměti a toto zpracování dat je řízeno instrukcemi. Instrukce je elementárním pokynem pro činnost procesoru. Procesor se obvykle skládá z komponent, kterými jsou aritmeticko-logické jednotky (ALU), řadiče, registry, sběrnice a další. Paměť, kterou procesor pro svoji činnost potřebuje, je určena pro data a instrukce. Jak je paměť rozčleněna a zda jsou data a instrukce ve stejné paměti společně, či nikoliv, je závislé na architektuře konkrétního procesoru. Posloupnost instrukcí označujeme jako program. Slouží jako vstup do procesoru spolu s daty potřebnými k jejich vykonávání [11, 25].

2.1.1 Klasifikace procesorů

Tato sekce klasifikuje procesory podle dvou kritérií. Jedná se o segment trhu, kde se uplatňují a o jejich mikroarchitekturu (vnitřní implementaci). Při sestavování klasifikace procesorů bylo čerpáno z [11].

Klasifikace procesorů podle segmentů trhu

- **Stolní**

  Jedná se o procesory použité ve stolních (angl. *desktop*) počítačích.

- **Mobilní**

  Do této kategorie spadají procesory určené do přenosných zařízení, to znamená, že se uplatňují v produktech napájených z baterií, a tudíž je žádoucí, aby spotřeba energie byla nižší, ale stále ne na úkor jejich výpočetního výkonu.

- **Vestavěné**

  Tento segment označuje procesory, jež jsou součástí vestavěných (angl. *embedded*) systémů, například může jít o auta, chytré spotřebiče, Internet věcí, zdravotní přístroje, apod. Pro takovéto procesory je důležitá nízká spotřeba energie, avšak toto většinou není omezující faktor, jelikož nevyžadují vysoký výpočetní výkon.

Klasifikace mikroarchitektur

- **Zřetězené/Nezřetězené**

  Zřetězené (angl. *pipelined*) procesory rozdělují zpracování instrukcí do více fází. Sled těchto fází se nazývá zřetězená linka (angl. *pipeline*). Výstup jedné fáze zřetězené linky je vstupem následující fáze. Vstupní instrukce prochází zřetězenou linkou sekvenčně a v konkrétním hodinovém taktu je tak v procesoru zpracováváno více instrukcí – každá v určité fázi rozpracovanosti.

  Zřetězené procesory se liší počtem fází, jimiž zpracování každé instrukce prochází (tj. délkou zřetězené linky). Takovéto procesory zřetězují zpracování instrukcí, či umožňují překrývání strojových instrukcí. To umožňuje využívat více částí procesoru paralelně a každá z těchto částí procesoru se věnuje přerozdělené příchozí sekvenci instrukcí určené přímo pro ni. Zřetězení (angl. *Pipelining*) [3] je efektivní technikou pro zvýšení výkonu a využívá se u všech procesorů. Nezřetězené procesory (angl. *non-pipelined*) se v praxi nevyskytují.

- **Seřazená/Neseřazená)**

  Procesor, založený na seřazené (angl. *In-Order*) mikroarchitektuře, zpracovává instrukce v pořadí, v jakém se objevují v programu a dle sémantiky instrukcí. Zatímco neseřazená (angl. *Out-of-Order*) je schopna vnitřně vykonávat instrukce i v jiném pořadí, než v jakém jsou uvedeny v programu, pokud to sémantika (datové závislosti mezi instrukcemi) dovolí. Důvodem pro zavedení neseřazeného zpracování je zvýšení výkonu, ovšem za cenu složitější hardwarové implementace procesoru.

- **Skalární/Superskalární**

  Skalární procesor dokončí nejvýše jednu instrukci v jednom hodinovém taktu. Superskalární procesor je schopen v jednom hodinovém taktu dokončit více než jednu instrukci, neboť je vybaven více výpočetními jednotkami schopnými pracovat paralelně.

- **Vektorová**

  Vektorové procesory disponují instrukcemi pro výpočty s vektory. V současnosti vektorové instrukce obvykle existují jako doplňky k základním instrukčním sadám procesorů.

- **Vícejádrové**

  Vícejádrový procesor se sestává z jednoho nebo více procesorových jader, z nichž každé disponuje vlastní sadou registrů a výpočetních jednotek a vykonává obecně vlastní sekvenci instrukcí (vlastní program).

- **Vícevláknové**

  Vícevláknový (angl. *multithreaded*) procesor je takový, který je schopen vykonávat více než jednu sekvenci instrukcí v rámci jednoho jádra. Zatímco vícejádrové procesory mají vyhrazené hardwarové prostředky, vlákna běžící na jednom jádře typicky většinu prostředků sdílí. Příkladem vícevláknových procesorů jsou proceosry firmy Intel s technologií *Hyper-threading*.

2.1.2 Registry procesoru

Registry jsou malá a velmi rychlá uložiště dat, do nichž si procesor ukládá data převzatá z operační paměti nebo taková, která jsou výsledkem výpočetních operací. Počet registrů, jejich druh a velikost je dána architekturou daného procesoru. Ne všechny zde uvedené druhy registrů musí být zastoupeny. Následující pojmy označují některé druhy registrů:

- **Střadač** (angl. *Accumulator*) je registr, který v sobě ukládá výsledek předchozí výpočetní operace. Vyskytuje se zejména ve velmi jednoduchých procesorech. V moderních procesorech je střadač zpravidla nahrazen všeobecnými registry.
- **Datový čítač** (angl. *Address register*) ukládá adresu na požadovaná data, je totiž efektivnější odkazovat se na data, než je ukládat.
- **Registr instrukcí** (angl. *Instruction register*) ukládá kód právě vykonávané nebo dekódované instrukce.
- **Programový čítač** (angl. *Program counter*) ukládá adresu do paměti, která odpovídá aktuálně vykonávané instrukci.

Existují další typy registrů, které jsou uvedeny a popsány například ve zdrojích [25, 32].

2.1.3 Vyrovnávací paměť

Vyrovnávací paměť procesoru (angl. *cache*) je rychlá paměť umístěná v blízkosti procesoru, má tedy nízkou latenci – dobu od požadavku na čtení nebo zápis dat po dokončení této operace. Tato paměť slouží k uchování instrukcí využívaných procesorem či dat načtených z operační paměti. Vyrovnávací paměť je první místo, kde procesor hledá potřebná data. V případě jejich nenalezení, procesor načte data z operační paměti a uloží je do vyrovnávací paměti.

Vyrovnávací paměti mohou být řazeny do více vrstev podle blízkosti procesoru. Používá se označení L1 až L4 [11], kde L1 je označení pro první vrstvu (angl. *layer*), která je nejblíže a má zpravidla velikost desítky kilobytů, je tedy i nejrychlejší. Procesor často mívá dvě paměti první vrstvy, kde v jedné jsou uložená data a v druhé instrukce. L2 a L3 má velikost mezi stovkami až tisíci kilobyty. Jedná-li se o vícejádrový procesor, vyrovnávací paměti s nízkou latencí (např. L1) bývají samostatné pro každé z jader a ostatní vyšší paměti jsou sdílené mezi jádry. V současnosti se využívají L3 a L4 spíše u výkonných počítačů (osobní počítače a servery), nikoli u vestavěných systémů.

2.1.4 Instrukční sada

Instrukční sada *ISA* (angl. *Instruction Set Architecture*), je rozhraní mezi procesorem a software vrstvou. Jedná se o popis fungování procesoru z pohledu programátora softwaru. Instrukční sada určuje množinu instrukcí, kterou procesor podporuje, počet a typ registrů

procesoru apod. Zkráceně řečeno, instrukční sada popisuje, co procesor umí [26]. Pojmy architektura a instrukční sada procesoru jsou úzce svázány a oba vystihují popis funkčnosti procesoru, jak je viditelný pro programátora software.

Instrukce je elementární pokyn, který je vykonáván procesorem. Každá taková instrukce nebo sekvence instrukcí má svůj binární kód a také textové označení v jazyce symbolických adres (angl. *Assembly language*). Jazyk symbolických instrukcí je nízkoúrovňový programovací jazyk, který je přímo překládán do binárního strojového kódu programu pro určitý procesor. Jazyk symbolických adres tedy není jen jeden univerzální jazyk, protože je specifický pro konkrétní architekturu procesoru, ale jedná se o označení pro druh jazyka. Překladač symbolických instrukcí je softwarový program nazývaný *assembler*, často se tak označuje i samotný jazyk.

Symbolické instrukce se skládají z klíčových slov, například STORE, LOAD, ADD, MUL a jiné [25]. Každá instrukce má definovanou její sémantiku a kolik přijímá na vstupu hodnot (operandů). Každá instrukce je před vykonáním dekódována – identifikován její typ a operandy – posléze je přeložena na základní operace zmíněné v předchozí větě a předána procesorovým jednotkám, určeným k těmto výpočtům. Příkladem může být instrukce ADD reprezentující operaci sčítání, která přijímá na vstupu dva operandy a její zápis v jazyce symbolických instrukcí vypadá následovně: *ADD EAX, 2*. Význam této instrukce je: přičtení hodnoty 2 k hodnotě v registru pojmenovaném *EAX*.

2.1.5 Modul pro ladění na čipu

Modul pro ladění na čipu (angl. *On-chip debug module*) je hardwarový subsystém procesoru, který umožňuje ladit program běžící na procesoru. Slouží k zastavení běhu procesoru a zkoumání, popřípadě změně, hodnot v registrech nebo dat v paměti. Umožňuje také aktivovat zarážky (angl. *breakpoint*) nebo *watchpoint* (mechanismy pro automatické zastavení procesoru při vykonání určité instrukce nebo při přístupu na uživatelem zadanou adresu v paměti). On-chip debug modul komunikuje vně procesoru, například přes rozhraní JTAG, popsané v sekci 2.6.

2.1.6 Duševní vlastnictví

Pojem duševní vlastnictví, nejčastěji používaný jako zkratka *IP* (angl. *Intellectual Property*), je v oblasti návrhu hardware často používán, proto je vhodné ho vysvětlit.

Průmyslové vlastnictví je součástí širšího souboru právních předpisů známých jako duševní vlastnictví. Obecně pojem duševní vlastnictví označuje právo nakládat s lidskými výtvory, které nemají hmotnou podobu (*know-how*, licence, návody, vynálezy apod.) [24]. Práva duševního vlastnictví chrání zájmy inovátorů a tvůrců.

V kontextu vývoje hardware se zkratka IP používá v užším smyslu k označení znovuvyužitelného návrhu hardware, který je možné formou licence poskytnout jinému subjektu. Pro takto poskytované bloky hardware se používá pojem IP jádra (angl. *IP cores*).

2.2 RISC-V

Projekt RISC-V vznikl v roce 2010 na americké univerzitě University of California, Berkeley za účelem výuky a výzkumu procesorových architektur. V žádném z prvních projektů RISC-V nebyly podány žádné patenty týkající se RISC-V, protože samotný RISC-V ISA nepředstavuje žádnou novou technologii. RISC-V ISA je založen na nápadech počítačové ar-

chitektury RISC, které se datují nejméně 40 let zpět. Implementace procesorů RISC (včetně některých jiných implementací otevřených standardů ISA) jsou široce dostupné od různých prodejců po celém světě [15].

V roce 2015 byla založena nezisková společnost RISC-V International[1] s cílem vybudovat otevřenou společnou komunitu inovátorů softwaru a hardwaru založenou na ISA RISC-V, jejímiž členy jsou i velké technologické firmy.

Celosvětový zájem o RISC-V je vysoký. Významným faktorem je to, že se jedná o bezplatný a otevřený standard, narozdíl od dnes dominantních architektur procesorů x86 a ARM. Standard umožňuje přenášet (angl. *portovat*) software, který umožňuje komukoli svobodně vyvíjet svůj vlastní hardware pro provozování nějakého softwaru [18]. RISC-V International nedodává žádné implementace procesorů RISC-V, pouze koordinuje a zaštiťuje vývoj specifikací architektury RISC-V. Podpora architektury RISC-V je nyní součástí mnoha softwarových projektů.

Využití procesorů s architekturou RISC-V je výhodné v tom, že výrobce takového řešení se nestává vázaným jen na jeden produkt od jedné společnosti, ale může si zvolit jiný produkt s ISA RISC-V bez nutnosti přepisovat software.

2.3 Abstrakce návrhu procesoru

Tato sekce vysvětlí pojem *syntéza* a popíše jednotlivé vrstvy abstrakce návrhu procesoru, aby bylo možné pochopit princip vývoje procesoru na jeho jednotlivých úrovních. Některé z úrovní abstrakce mohou představovat implementaci návrhu v podobě kódu. Takový kód lze s pomocí dodatečného softwaru simulovat a testovat. Z těchto zdrojů [8, 12, 37, 39], které popisují nejen dané téma, ale uvádí i praktické a konkrétní použití, je čerpáno v této sekci.

2.3.1 Syntéza

Během procesu návrhu je integrovaný obvod (procesor) obvykle nejprve specifikován pomocí abstrakce vyšší úrovně. Implementaci obvodu pak lze chápat jako hledání funkčně ekvivalentního vyjádření na nižší úrovni abstrakce. Pokud je hledání provedeno automaticky pomocí softwaru, použije se termín syntéza. Syntéza je tedy automatická konverze reprezentace obvodu na vysoké úrovni na funkčně ekvivalentní reprezentaci obvodu na nízké úrovni. Jinak řečeno, syntéza označuje transformaci popisu hardwaru z vyšší úrovně na nižší.

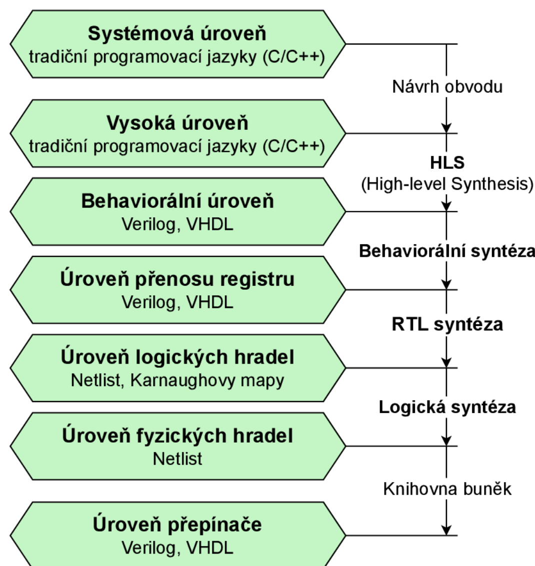
Logická syntéza

Logická syntéza je proces, který probíhá při přechodu z RTL na úroveň hradel nebo tranzistorů. Výstupem procesu syntézy na této úrovni je soubor *netlist*, který se používá jako vstup do dalších nástrojů. Netlist je obvykle textový soubor nebo může být vizualizován ve formě schématu, které pomůže vizualizovat digitální obvod. Logická syntéza je široce používána v aplikačně specifických integrovaných obvodech (*ASIC*) a na deskách založených na FPGA.

[1] `https://riscv.org/`

Fyzická syntéza

Fyzická syntéza je proces převodu netlistu nebo RTL popisu do umístěného a propojeného netlistu (častěji používaný anglický výraz *placed and routed netlist*) bez nutnosti interakce uživatele v mezilehlých krocích toku návrhu. Fyzická syntéza je obecně podporována pouze nástroji vyšší třídy, protože algoritmy jsou složitější a vyžadují podrobnou znalost cílové technologie obvodu (FPGA nebo ASIC).



Obrázek 2.1: Přehled úrovní abstrakce digitálních obvodů a přechody mezi nimi s pomocí určitých druhů syntéz

2.3.2 Úrovně abstrakce návrhu procesoru

Jak pro návrh, tak pro testování procesoru se využívají různé úrovně abstrakce (Obrázek 2.1).

Systémová úroveň (angl. *System Level*)

Abstrakce na úrovni systému je zaměřena pouze na jeho nejvýš (abstraktně) postavené stavební jednotky, jako jsou CPU a výpočetní jádra. Na této úrovni je obvod obvykle popsán pomocí tradičních programovacích jazyků, jako je jazyk C nebo C++ či Matlab. Někdy se používají speciální softwarové knihovny, které jsou zaměřeny na simulační obvody na úrovni systému, například SystemC.

Vysoká úroveň (angl. *High level*)

Abstrakce systému na vysoké úrovni (někdy označovaná jako algoritmická úroveň) je také často představována pomocí tradičních programovacích jazyků, ale se sníženou sadou funkcí. Existují nástroje [23] pro syntézu kódu na vysoké úrovni *HLS* (angl. zk. *High-level synthesis*), obvykle ve formě kódu C, C++ nebo SystemC s dalšími metadaty, na behaviorální kód *HDL* (angl. zk. *Hardware Description Language*), konkrétně například jazyk Verilog či VHDL.

Behaviorální úroveň (angl. *Behavioural Level*)

Na behaviorální úrovni se k popisu obvodu používají jazyky zaměřené na popis hardwaru, jako je Verilog nebo VHDL, přičemž alespoň v části popisu obvodu se používá tzv. *behaviorální modelování*. Kód behaviorální úrovně může být použit k popisu procesoru, který je určený k syntetizování.

Úroveň přenosu registru (angl. *Register-Transfer Level*)

Jde o základní abstrakci používanou pro definování elektronických systémů. Návrh RTL je obvykle zachycen pomocí jazyka popisu hardwaru (HDL), jako je Verilog nebo VHDL, přesněji řečeno pomocí tzv. syntetizovatelné podmnožiny těchto jazyků. To znamená, že jsou využity takové konstrukce jazyka, které jsou přenositelné do navazujících nástrojů logické syntézy [34]. Podrobněji se práce RTL věnuje v následující sekci 2.4.

Úroveň logických hradel (angl. *Logical Gate Level*)

Na úrovni logických hradel je design reprezentován tzv. *netlistem*, který používá pouze buňky z malého počtu jednobitových buněk, jako jsou základní logická hradla (AND, OR, NOT, XOR, atd.) a registry. Zde musí logická syntéza řešit dva problémy. Nejprve najít příležitosti pro optimalizaci v rámci netlistu na úrovni hradel a za druhé optimální (nebo alespoň dobré) mapování netlistu logických hradel na ekvivalentní netlist fyzicky dostupných typů hradel [5].

Úroveň fyzických hradel (angl. *Physical Gate Level*)

Na úrovni fyzických hradel se používají pouze hradla, která jsou fyzicky dostupné v cílové technologii. V některých případech to mohou být pouze hradla NAND, NOR a NOT a také registry typu D. V ostatních případech to může zahrnovat buňky, které jsou složitější než buňky použité na úrovni logických hradel. V případě návrhu založeného na FPGA je reprezentace úrovně fyzického hradla v podobě netlistu reprezentovaného vyhledávací tabulkou *LUT* (angl. zk. *Lookup table*) s volitelnými výstupními registry, neboť toto jsou základní stavební prvky logických buněk FPGA. Pro syntézu nástrojů je tato abstrakce obvykle nejnižší úrovní.

Úroveň přepínačů (angl. *Switch Level*)

Reprezentace obvodu na úrovni přepínače (neboli tranzistorů), je netlist využívající jednotlivé tranzistory jako buňky. Modelování na úrovni přepínačů je možné ve Verilogu a VHDL, ale v moderních designech se používá jen zřídka, protože v moderních digitálních ASIC nebo FPGA zařízeních jsou fyzická hradla považována za atomické bloky logického obvodu.

2.4 RTL

Zkratka RTL má více významů v oblasti designu obvodů, které jsou níže uvedeny [8].

1. V oboru tvorby překladačů znamená zkratka RTL registrační jazyk přenosu (angl. *Register transfer language*). Používá se k popisu toku dat mezi architekturálními registry, popsané v podsekci 2.3.2. Pro tuto práci je tento význam zkratky RTL nepodstatný.

2. Pro návrháře digitálních obvodů znamená RTL úroveň přenosu registru (angl. zk. *Register-transfer level*), což je pojem označující konkrétní úroveň abstrakce popisu digitálních obvodů. Na této úrovni abstrakce se typicky používají jazyky pro popis hardware jako Verilog nebo VHDL. Obvod je popsán jako sada registrů propojená pomocí kombinačních logických prvků (hradel).

Je-li v následujícím textu práce použitá zkratka RTL, myslí se tím popis hardware na úrovní abstrakce meziregistrových přenosů, jak byla popsána v sekci 2.3.2. Návrh v reprezentaci RTL je obvykle zapsán pomocí HDL, jako například podmnožina jazyků Verilog, SystemVerilog nebo VHDL [8]. Použití HDL na této úrovni je výhodné pro simulaci, protože k simulaci obvodů popsaných na úrovni RTL je možné provádět přímo, bez dalších nástrojů. Mnoho optimalizací a analýz lze nejlépe provádět na úrovni RTL [2, 4, 13, 16].

Jazyk VHDL definuje [2, 8] RTL jako úroveň popisu digitálního návrhu, ve které je taktované (časované) chování návrhu výslovně popsáno. Tedy pokud jde o datové přenosy mezi úložnými prvky v sekvenční logice a té kombinační, která může představovat jakýkoli výpočet nebo logiku aritmeticko-logické jednotky. Modelování RTL umožňuje sestavit hierarchii návrhů, která představuje strukturální popis ostatních modelů RTL.

RTL ve smyslu kódu pro simulování může využívat kteroukoli z dostupných funkcí jazyka. Toto je často přístup, když kód není určen pro syntézu do FPGA. Plně funkční HDL kód však lze použít pro abstraktní, algoritmické modelování finální funkčnosti FPGA, které má návrh nakonec vytvořit. Paralelismus poskytovaný HDL někdy poskytuje přirozenější způsob vyjádření funkčnosti, než je možné v čistě sekvenčních jazycích, jako je C/C++.

RTL je založen na synchronní logice a obsahuje tři hlavní části: registry, které obsahují informace o stavu, kombinatorickou logiku, která definuje vstupy stavu vnoření a hodiny, které řídí, kdy se stav registrů změní.

2.4.1 RTL simulatory

Simulace je obecný termín pro postup, kdy pomocí počítačového software napodobujeme chování jiného systému (v tomto případě digitálních obvodů). Používá se ke studiu vztahů mezi vstupními parametry systému a jeho chováním. Pomocí simulace je například možné zjistit chyby, které způsobí nesprávnou reakci návrhu. V ostatních případech umožňuje optimalizaci návrhu pro maximální výkon nebo ekonomiku provozu nebo stavby. Popřípadě může být systém tak složitý, že simulace je jediným způsobem, jak lze ovládat a studovat proměnné ovlivňující návrh procesoru a jejich vzájemné chování [22]. Existují různé metody [4, 22] používané k simulaci digitálních logických obvodů za účelem predikce jejich chování za přítomnosti různých podnětů a faktorů prostředí, ve kterých budou obvody využity.

Pod termínem simulace budeme v práci rozumět simulaci digitálních obvodů, popsaných na úrovni RTL, prostřednictvím některého z nástrojů pro simulaci digitálního hardware. (*verifikace*) a ladění (angl. *debugging*) obvodů popsaných na úrovni RTL [16] je složitým technickým problémem, a to z těchto důvodů:

- Formální ověření není škálovatelné, a proto je možné ho použít zpravidla jen pro omezené podsystémy, nikoli pro celý systém.

- Softwarová simulace je příliš pomalá, protože na rozdíl od skutečných digitálních obvodů probíhá sekvenčně, nikoli paralelně. Je také velmi těžké generovat vstupní sady, které během simulace softwaru dostatečně otestují jeho funkcionalitu.

- Rychlé prototypování pomocí čipů FPGA omezuje ovladatelnost a viditelnost. V případě nalezení chyby je u FPGA protypu obtížné zjistit její příčinu, nelze totiž sledovat vnitřní signály designu a vývoj jejich hodnot v čase.

Konečným cílem simulace návrhu před výrobou hardwaru je odstranit všechny chyby návrhu. To je žádoucí, aby se zabránilo nutnosti znovu vyrábět hardware a zajistilo se, že finální hardware funguje správně. Vzhledem ke složitosti stále větších návrhů obvodů je praktičtější dostatečně simulovat dosažení plně funkčního prototypu. Nejhorším scénářem je mít první vyrobený hardware při zapnutí ve stavu, který vylučuje vlastní testování.

Pravděpodobnost a náklady na chyby návrhu obecně rostou se složitostí. Chyby návrhu zjištěné po velkém počtu produktů dodaných na trh mohou vést k neúnosným nákladům. V jednom slavném příkladu se říká, že aritmeticko-logická chyba návrhu stála společnost podle [13] půl miliardy dolarů. Poznamenává také, že chyby v návrhu mohou mít katastrofické účinky i u produktů menšího rozsahu. Příkladem jsou starty a nesprávná funkce satelitů za miliardu dolarů a životně důležité řídicí jednotky systému v nemocnicích. Chyby návrhu nalezené zákazníkem před vydáním jsou také velmi nákladné, což vede k dlouhým cyklům regresního testování, neschopnosti provést vlastní test na fyzicky vyrobeném procesoru, jeho velké odezvě, a nakonec zpoždění doby uvedení na trh. Naštěstí lze chyby zjištěné na začátku návrhu konstrukčního cyklu opravit mnohem snadněji a za nižší cenu než ty, které byly zjištěny později. Tyto zkušenosti přinutily projektové manažery a konstruktéry zaměřit více pozornosti na všechny aspekty ověřování. Proto je výkonnost návrhových a analytických nástrojů v každé ověřitelné metodice RTL nesmírně důležitá.

Na trhu jsou k dispozici různé nástroje pro simulaci designu popsaného na úrovni RTL. Kromě samotné RTL simulace mohou mít tyto nástroje i jiné doplňkové funkce. Jedná se například o tyto nástroje:

- Aldec — Riviera Pro,
- Metric Technologies — Metrics Cloud Simulator,
- Cadence — Incisive, XCelium,
- Mentor Graphics — ModelSim, Questa,
- Synopsys — VCS,
- Xilinx — ISE Simulator (ISim).

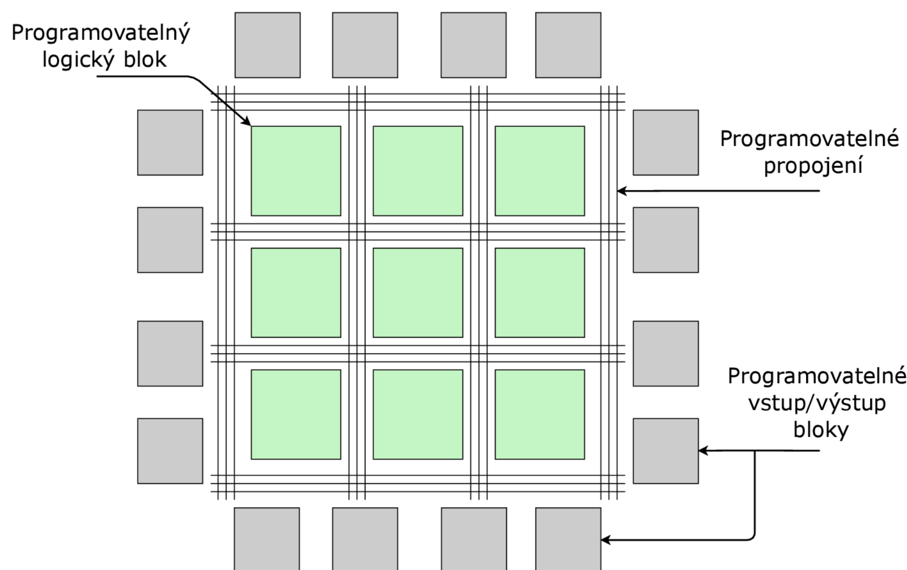
Obvyklé licenční poplatky se pohybují řádově v deseti tisících až stovkách tisíc dolarů za měsíc[2]. Už jen evaluace takovýchto simulátorů může být velmi nákladná. Je zde tedy i další možnost, a to uplatnit pro testování rychlé prototypování pomocí čipů FPGA pro konkrétní způsob testování a tímto způsobem provádět některé formy testů, u kterých je to možné. Po finanční stránce je tento přístup výhodný, neboť nevznikají náklady za licence RTL simulátorů a stávající licence nejsou obsazeny kvůli testování. Je možné je využít i pro jiné účely. Desky FPGA jsou představeny v další sekci.

Existují i simulátory, které jsou k dispozici zdarma[3]. Takové simulátory na složitějších obvodech můžou mít dlouhou dobu trvání simulace nebo nenabízí funkcionalitu, která je u komerčních simulátorů dostupná [31].

[2] Ceny jsou obvykle smluvní a předmětem obchodních jednání.
[3] Verilog: https://www.veripool.org/ a Icarus: http://iverilog.icarus.com/

2.5 FPGA

FPGA (angl. zk. *Field-Programmable Gate Array*) je polovodičové zařízení (digitální integrovaný obvod), jehož funkce není z výroby pevně daná, ale obvod je konfigurovatelný zákazníkem. Po jeho výrobě je připraveno ke konfiguraci, kde se k jeho nakonfigurování používá soubor nazývaný *bitstream*, pomocí kterého se zařízení naprogramuje a obvod FPGA začne plnit funkci konkrétního, uživatelem specifikovaného digitálního obvodu. Zařízení je založeno na matici konfigurovatelných logických bloků připojených prostřednictvím programovatelných propojení a vstupních/výstupních bloků zastupujících rozhraní pro logické bloky do vnějších komponent v FPGA [8, 16, 38]. Kromě konfigurovatelných logických bloků mohou FPGA čipy už z výroby obsahovat další prvky, například paměti, komunikační rozhraní a podobné. Vnitřní struktura FPGA se liší u jejich jednotlivých výrobců a produktových řad. Zde na obrázku 2.2 je uveden obecný příklad.

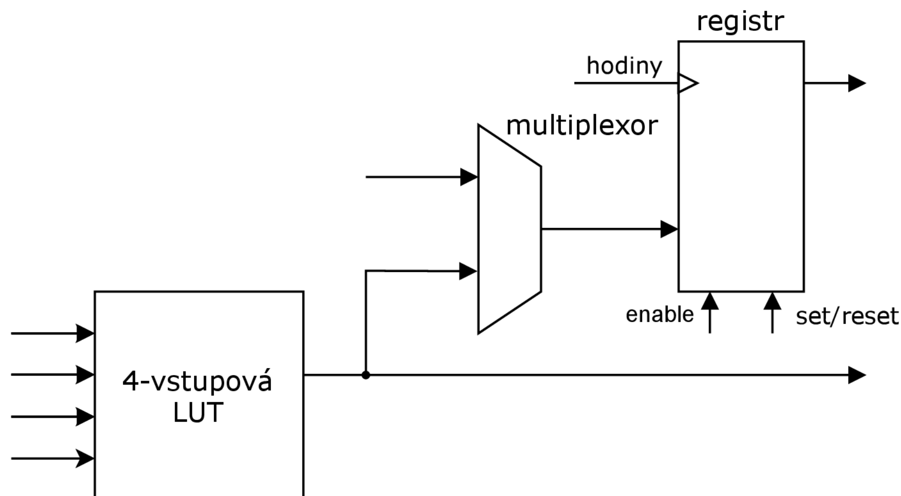


Obrázek 2.2: Obecné schéma FPGA

- **Konfigurovatelné logické bloky** (angl. *Configurable Logic Blocks*) jsou prostředky pro implementaci logických funkcí. Každý logický blok se skládá z vyhledávacích tabulek (angl. *Look-up tables*), multiplexorů a registrů. Všechny tyto elementy jsou programovatelné. Registry je možné nakonfigurovat tak, aby implementovaly klopný obvod (Obrázek 2.3). Existují však i logické bloky implementující vyhledávací tabulky s více vstupy [8, 19].

- **Programovatelná propojení** (angl. *Programmable Interconnect*) poskytují pro logické bloky vyhrazené cesty s minimálním zpožděním (např. hodinový signál) a umožňují spojení mezi různými logickými a vstupními/výstupními bloky.

- **Programovatelné vstupní/výstupní bloky** slouží jako média či prostředek pro rozhraní logických bloků a směrovacích architektur do vnějších segmentů FPGA. Primární výstupy/vstupy někdy označované jako GPIO (angl. *General-Purpose Input/Output*) [29].



Obrázek 2.3: Obecné schéma struktury konfigurovatelného logického bloku

FPGA čip může být pomocí svých vnitřních programovatelných prvků překonfigurován na uživatelem určený digitální obvod (například procesor). Tudíž před výrobou nemusí být známa celá specifikace budoucího využití desky FPGA. Tímto se FPGA liší od specificky vyrobeného integrovaného obvodu pro určitou aplikaci *ASIC* (angl. zk. *Application Specific Integrated Circuits*), přičemž existují varianty FPGA, které jsou po výrobě volitelně programovatelné jen jednou [14, 20].

Některá FPGA, obvykle označované jako SoC (angl. zk. *System on a chip*), obsahují jeden nebo více jader procesorů přímo na desce. Některé architektury FPGA desku rozšiřují o integrované SRAM bloky nazývané bloková RAM (BRAM), fázový závěs (PLL) nebo správce hodin. Lze také přidat bloky digitálního zpracování signálu (DSP), které obsahují konfigurovatelné multiplikátory a konfigurovatelný sčítač, umožňující provádět vícenásobné operace (MAC). Ne všechny FPGA tyto rozšířené funkce mají, záleží na jejich zaměření.

Konfigurace logických bloků a způsobu propojení uvnitř FPGA je určeno jednotlivými bity souboru bitstream nahraného do čipu FPGA. Paměťové buňky, které uchovávají jednotlivé bity bitstreamu, výrobci FPGA obvykle implementují jednou z těchto tří technologií:

- **Antifuse** je technologie, kde jsou konfigurovatelné buňky programovatelné jen jednou. Po prvním programování tedy čip FPGA nelze přeprogramovat. Slouží k specifickým bezpečnostním aplikacím. Zařízení s takovou technologií se nevyrábí mnoho, proto je jejich cena vysoká.

- **Flash** technologie, stejně jako Antifuse, má buňky energeticky nezávislé, avšak je možné je přeprogramovat. Konfigurační buňky jsou odolné vůči záření, to zajišťuje jejich využití v aplikacích pro kosmické programy.

- **SRAM** tato technologie uchovává konfigurační bitstream prostřednictvím klopných obvodů. Výhodou je, že je možné konfiguraci měnit bez omezení. Nevýhodou je, že konfigurace je ztracena v okamžiku, kdy je od čipu FPGA odpojeno napájení. Proto musí být konfigurační bitstream uložen v jiné trvalé paměti a odsud do čipu FPGA nahrán při každém zapnutí. Existují i jiné dynamické konfigurační scénáře, než je zmíněno.

Nejčastěji se setkáváme s variantou založenou na SRAM, protože umožňuje opakované přeprogramování, v tomto případě, v průběhu vývoje designu procesoru.

Možnosti využití FPGA

FPGA zařízení se využívají v produktech a zařízeních v mnoha různých oborech [14]. Některé příklady využití v praxi jsou: automotivní IP řešení na bázi křemíku pro asistenční systémy v autech, diagnostika a zobrazování potřebných dat v lékařských oborech, řízení motoru, sledování komunikace na počítačových sítích, strojové vidění a učení, umělá inteligence, těžba a vyhodnocování dat v datových centrech, zpracování signálů ve vojenských rádiových stanicích a zpracování zvuku či obrazu.

FPGA čipy se uplatňují v oblastech, kde lze s výhodou uplatnit změna konfigurace (například při aktualizaci systému na novější verzi) nebo při sériové nebo kusové výrobě zařízení, u níž by výroba aplikačně specifického integrovaného obvodu (ASIC) byla neekonomická.

Výhodu způsobující takto široké pole použití FPGA lze vysvětlit na následujícím příkladu z praxe. Uvažujme komplexnější systém, který se bude časem potřebovat přizpůsobit nejnovějším technologiím – kamerový systém rozpoznávající obličeje. Objeví se nová technologie kamerového senzoru, který ale využívá jiný komunikační protokol než ten dosavadní. Aplikování nové technologie by bylo možné jen za předpokladu výměny celé části systému, která umí komunikovat jen pomocí starého protokolu. Při využití FPGA lze vyměnit jen senzor a FPGA zařízení přeprogramovat, aby podporovalo nový protokol.

Největší firmy zabývající se výrobou FPGA jsou: Altera, Intel, Lattice Semiconductor, Microchip, Xilinx. Dva hlavní producenti FPGA zařízení jsou Intel s podílem trhu 35% (po akvizici firmy Altera) a Xilinx 52% k roku 2019 [7]. Obě firmy mají široké portfolio nabízených produktů, od méně výkonných (angl. *low-end*) po velmi výkonné (angl. *high-end*) [19].

Pro účely této práce je využito FPGA zařízení od firmy Xilinx – čip XC7A100T řady Artix-7, osazený na vývojové desce Nexys A7 od firmy Digilent [6].

2.6 JTAG

JTAG [30, 36] je standard pro komunikaci s integrovanými obvody (například procesory), který slouží k testování propojů na plošných spojích, k interních funkcí obvodů a také pro konfiguraci procesorů, FPGA, flash pamětí a podobných komponent. Standard umožňuje přistupovat k digitálním rozhraním v zařízení a jedná se o jeden z hlavních prostředků pro testování vestavěných systémů. Rozhraní JTAG je běžnou součástí programovatelných digitálních obvodů jako jsou čipy FPGA nebo procesory.

Rozhraní JTAG je z hlediska této práce významné ze dvou důvodů:

- Nahrání konkrétního hardwarového obvodu (v podobě tzv. bitstreamu) do čipu FPGA se realizuje prostřednictvím rozhraní JTAG.
- Interakce s testovaným procesorem pro potřeby ladění (angl. *on-chip debugging*) se provádí také prostřednictvím rozhraní JTAG. Rozhraní JTAG testovaného procesoru je oddělené a nezávislé na JTAG rozhraní FPGA čipu, uvnitř je procesor implementován.

Pro komunikaci prostřednictvím rozhraní JTAG je třeba:

- Hardwarový adaptér, který konvertuje elektrické signály rozhraní JTAG na některé z běžných rozhraní osobního počítače, obvykle USB. Pro tyto hardwarové adaptéry se také používají termíny *JTAG dongle*, *JTAG probe* a podobné.
- Softwarový program, který umožní provádět akce s daným hardwarovým zařízením, například čtení nebo zápis do paměti.

Jako hardwarový adaptér je v rámci této práce použit produkt JTAG-HS2[4] od firmy Digilent. Jako softwarový program pro ovládání hardwarových zařízení prostřednictvím JTAG je využit nástroj OpenOCD[5].

[4] https://store.digilentinc.com/jtag-hs2-programming-cable/
[5] http://openocd.org/

Kapitola 3

Testování procesorů

Procesor, jako každý jiný produkt, během vývoje prochází testovací fází, kterou se tato kapitola zabývá. Cílem fáze testování návrhu procesoru vůči specifikaci – používá se termín verifikace. Procesor je složitý synchronní digitální obvod. K jeho verifikaci se tedy používají metody vhodné pro tento typ obvodů. Kapitola popisuje a srovnává dva základní přístupy k verifikaci hardware – funkční verifikaci a formální verifikaci [21, 27].

3.1 Funkční verifikace

Funkční verifikace (angl. *Functional verification*) ověřuje, zda model obvodu plní specifikaci a to sledováním jeho vstupů a výstupů, a to nejčastěji prostřednictvím simulace. Jak už bylo dříve zmíněno, není ekonomicky ani technicky reálné vyrábět skutečné fyzické prototypy (polovodičové čipy) za účelem testování návrhu procesoru. Takovýto přístup by poslal cenu za testování procesorů do astronomických čísel a stejně tak i dobu, kterou by vývoj procesoru zabral. Podle některých zdrojů verifikace při vývoji ASIC zabírá v průměru něco přes 50% času [10] a přibližně 40% prostředků v projektech zaměřených na návrh digitálních obvodů je použito na funkční verifikaci [21]. Funkční verifikace může být zobecněna do cyklu ve 4 bodech:

1. Vývoj obsahující plán verifikace, návrh verifikační architektury *testbench* a tvorba testů.
2. Simulace: softwarová, akcelerační, emulační a další.
3. Ladění (debug) zvolených úrovní abstrakce.
4. Pokrytí: funkční, kódu nebo pomocí SVA [35] (angl. zk. *SystemVerilog Assertions*), což vrací cyklus k první bodu cyklu.

Ověření pomocí funkční verifikace musí být provedeno ještě před zahájením výroby fyzických čipů. Anglický termín *tape-out* označuje stav vývoje, kdy procesor přechází do fáze výroby. Platí, že procesor před tape-out by měl nutně mít návrh procesoru ověřen. Verifikace je považována za ukončenou ve chvíli, kdy je dosažená určitá úroveň pokrytí (angl. *coverage*), protože vstupní procesy ve verifikaci typicky nepokryjí celý stavový prostor verifikované jednotky. S detailnějším popisem návrhu roste čas a výpočetní složitost vyžadována pro verifikaci. Minimální požadovaná úroveň pokrytí je určena návrhářem a je

součástí specifikace. Proto funkční verifikace sice nedosahuje síly formální verifikace (neposkytuje záruku korektnosti), ale na druhou stranu není tak časově a výpočetně náročná a netrpí stavovou explozí.

V této práci se omezíme na funkční verifikaci procesorů na abstrakční úrovni RTL, kde se její popis sestává z hardwarových registrů a kombinační logiky, která registry popisuje. Důvody pro vymezení jsou následující:

- Popis procesoru na úrovni RTL je produkt, který firma Codasip dodává zákazníkům.
- Výhody RTL popisu jako je možnost ho studovat, upravovat, ladit a simulovat. Navíc je použitelný v nástrojích různých výrobců (je tzv. *portabilní*).

Následně jsou uvedeny metody funkční verifikace na úrovni RTL.

3.1.1 Softwarová simulace

Při softwarové simulaci je důležité si uvědomit že procesy jsou prováděné sekvenčně – jsou řízeny událostmi (angl. *event-driven*). Oproti tomu hardware vykonává ze své podstaty veškerou činnost paralelně. To znamená, že softwarová simulace je nutně pomalejší. Tato simulace se provádí v softwarových simulátorech popsaných v sekci 2.4.1. Když srovnáme výkon simulace RTL oproti simulaci architekturálního popisu, tak se výkon snižuje až 5násobně (vyšší složitost), ale odhalí se nejvíce funkčních chyb.

Principem softwarové simulace je přikládání testovacích hodnot na vstupy obvodu (procesoru), které jsou označované jako *stimuly*. Stimuly mohou být ručně vytvořené, pseudonáhodně generované nebo také kombinace obojího. Následně přichází na řadu kontrola očekávaných výstupů – ruční nebo automatická. Kvalitou verifikace prostřednictvím simulace je možné posuzovat pomocí měření pokrytí (angl. *coverage*).

Výhody

- Softwarová simulace umožňuje zobrazit hodnotu některého signálu nebo registru uvnitř procesoru v libovolném okamžiku v průběhu simulace, v libovolném hodinovém taktu, a to je umožněno v reálném čase nebo zpětně po skončení simulace. Signály se typicky zobrazují v časových diagramech – tzv. vlnky (angl. *waveforms*). Jinými slovy, jedná se o tzv. *white-box* testování.
- Funguje i pro velmi složité systémy.
- Zajišťuje reprodukovatelnost chyb (tj. softwarová simulace je deterministická).

Nevýhody

- Sekvenční způsob simulování v softwaru je důvodem nízké výkonnosti (oproti tomu činnost hardwaru na čipu probíhá paralelně).
- Vysoká cena za licence pro simulační nástroje.

3.1.2 Softwarová simulace s hardwarovou akcelerací

Principem této metody je, že část výpočtů v rámci simulace je provedena ve specializovaném hardwaru. Akcelerační hardware má podobu přídavných hardwarových karet (proprietární

řešení, často nákladné), obvykle založeno na výkonných obvodech FPGA. Typicky se jedná o software v podobě proprietárního komerčního řešení podporující jen nějaký konkrétní hardware. Váže se také na konkrétní RTL simulátor a rozšiřuje jeho funkčnost. Jedním z takových existujících řešení je Aldec HES-DVM, které má společné jen některé rysy. Produkt je popsaný v sekci 4.4.

Výhody

- Rychlejší než čistá softwarová simulace.

Nevýhody

- Vysoká cena za licence a akcelerační hardware.

3.1.3 Rychlé prototypování pomocí FPGA

Na metodu funkční verifikace pomocí rychlého prototypování pomocí FPGA se zaměřuje tato práce. Popis procesoru v podobě RTL se pomocí syntézy a procesu *place and route* [21] převede na tzv. bitstream. Poté se bitstream nahraje do FPGA a tímto se získá prototyp procesoru, který je možné použít ke spuštění některých testů. Place and route je proces, který má dvě fáze. První fáze (angl. *place*) je rozhodování, kam umístit všechny elektronické součástky, v rámci matice konfigurovatelných bloků uvnitř čipu FPGA. Druhou fází je propojování (angl. *route*), které rozhodne o přesném návrhu všech vodičů potřebných pro připojení a propojení umístěných komponent. Proces převodu RTL reprezentace procesoru na bitstream se provádí pomocí softwarových nástrojů od výrobce čipu FPGA. V případě FPGA od firmy Xilinx se jedná o nástroj Xilinx Vivado[1]. Testy vhodné pro testování procesoru v FPGA jsou popsány v sekci 4.2.

Výhody

- Nízké náklady.
- Vysoká rychlost.

Nevýhody

- Není přístup k vnitřním signálům a registrům procesoru, to je hlavní rozdíl oproti dříve zmíněné softwarové simulace v sekci 3.1.2. Jedná se tedy o *black-box* testování.
- Doba trvání převodu digitálního návrhu ve formě RTL do bitstreamu (syntéza a place and route)

Termín *black-box* značí způsob testování na základě specifikací požadavků a kontroluje správnost očekávaných výstupů, nezkoumá se vnitřní logika implementace a její chování na nízké úrovni. Oproti tomu *white-box* zkoumá správnost implementovaného kódu, v souvislosti s návrhem procesoru se jedná o zkoumání například správnosti hodnot jednotlivých vnitřních signálů a registrů. V případě testování pomocí prototypů založených na obvodech FPGA se vždy jedná o black-box testování, neboť vnitřní signály procesoru nejsou dostupné k inspekci.

[1] https://www.xilinx.com/products/design-tools/vivado.html

Převod digitálního návrhu do podoby bitstreamu je jednorázová operace – pro daný RTL design se provede jen jedenkrát. Posléze je možné daný bitstream opakovaně používat pro různé testy. Proces generování bitstreamu je proprietální, musí se tedy použít softwarové nástroje od výrobce FPGA čipu.

Rychlé prototypování v FPGA je možné výhodné kombinovat s metodou softwarové simulace. Testy, u kterých je to technicky možné (u nichž nevadí přístup typu black-box), se provedou s využitím prototypu pomocí FPGA. V případě selhání testu je možné provést ladění testů prostřednictvím simulace, která inženýrům dává možnost detailně zkoumat vnitřní registry a signály obvodů, a tím odhalit přesné místo výskytu chyby.

3.2 Formální verifikace

Formální verifikace (angl. *Formal verification*) není pro tuto práci stěžejní. Je zde uvedena, protože se jedná o jiný přístup k verifikaci než verifikace funkční. Oba tyto přístupy se v praxi obvykle kombinují. Často se formální metody používají jako doplňkové verifikační přístupy k softwarové simulaci, popřípadě pro verifikaci menších částí složitějších obvodů, například podsystémů procesoru.

Principem formální verifikace [33] je ověřit pomocí formálních matematických metod, zdali obvod splňuje určitou vlastnost. Výsledkem formální verifikace je důkaz o správnosti nebo protipříklad. Důkazem správnosti se rozumí, že neexistuje žádný vstup, který by způsobil porušení sledované podmínky. Protipříklad je určitý běh systému nebo jeho výstup, který vede k porušení sledované podmínky. Existuje celá řada různých formálních metod. Níže jsou uvedeny dva příklady:

Kontrola logické ekvivalence (angl. *Logic equivalence checking*) je důkaz toho, že dvě různé implementace jsou ekvivalentní – tzn. že popisují stejný obvod. Příkladem konkrétního softwaru je nástroj SLEC[2] ze softwarového produktu Questa Formal.

Kontrola modelu (angl. *Model Checking*) ověřuje vlastnosti systému úplným prozkoumáním jeho stavového prostoru. Verifikovaný systém je typicky reprezentován konečným automatem nebo jeho variantami.

Výhody

- Formální metody poskytují matematický důkaz správnosti obvodu. V případě chybného obvodu poskytují protipříklad.

Nevýhody

- Problém stavové exploze – výpočetní náročnost formálních metod výrazně roste s rostoucí složitostí verifikovaného obvodu. Není tedy vhodná pro sekvenční obvody a obvody zpracovávající velké objemy dat (velký stavový prostor).

- Nutnost formálního zadání – podmínka, kterou kontrolujeme, musí být také formálně popsána

[2] https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/slec/

Kapitola 4

Analýza

V této kapitole se práce zaměřuje na konkrétní druhy testů spouštěnými nad procesory vyvinutými ve firmě Codasip, popsanými na úrovni RTL. Analyzuje princip a strukturu těchto testů a také zdůvodňuje, které testy je možné provádět pomocí rychlého prototypování s využitím čipů FPGA. Tento proces prototypování však není zahrnut do aktuálního firemního testovacího systému, což se firma rozhodla změnit. Jsou tak zde uvedeny důvody, které vedly k implementaci systému jakožto výstupu této práce a také k jeho cílovému využití.

4.1 Aktuální stav

V současné době je veškeré automatické testování integrované v testovacím systému a provádí se pomocí softwarových simulátorů. Využití softwarových simulátorů vede k velkým výdajům za licence a k zvyšujícím se potřebám nákupů dalších výpočetních zdrojů, kde simulování a testování procesorů probíhá. Je tedy potřeba zjistit a zvážit, u kterých testů je možné aplikovat rychlé prototypování v FPGA.

V současnosti jsou desky FPGA ve firmě Codasip využívány tak, že se přiřadí zaměstnanci, který s deskou pracuje formou, že ji připojí k firemnímu počítači. Při neustálém fyzickém přesouvání jednotlivých desek se desky mohou poškodit. Při přepojení k počítači, kam byla deska připojena poprvé, je nutností připravit prostředí a instalovat potřebný software pro zajištění správné komunikace desky s počítačem. Při instalaci ovladačů a softwarového vybavení pro práci s vývojovou deskou můžou vždy nastat neočekávané komplikace.

4.2 Testy reprezentace RTL procesorů firmy Codasip

Tato sekce přiblíží druhy testů v současnosti používané firmou Codasip. Dále uvede, u kterých je testování v FPGA výhodně či vůbec možné.

4.2.1 Verifikace založená na UVM

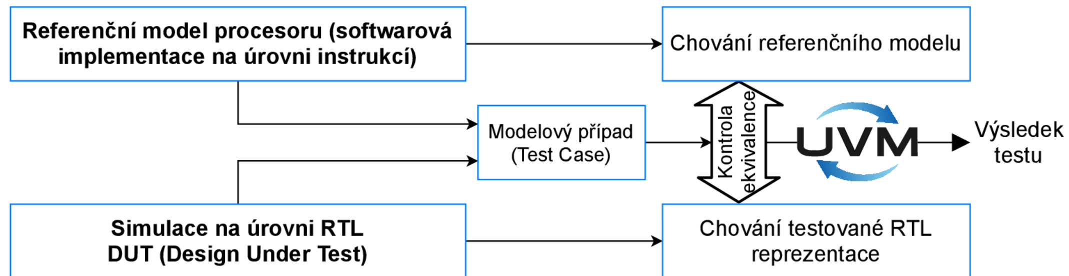
UVM (angl. *Universal Verification Metodology*) je metodologie pro verifikaci digitálního hardwaru standardizovanou organizací Accelera[1] v roce 2011. Jedná se také o stejnojmenou dynamickou knihovnu napsanou v jazyce SystemVerilog, její implementace je open-source[2].

[1] https://www.accellera.org/community/uvm

[2] https://www.accellera.org/downloads/standards/uvm

Knihovna umožňuje vytváření opakovaně použitelných ověřovacích komponent a sestavování testovacích prostředí s využitím omezeného generování náhodných stimulů a metodik funkčního pokrytí (*coverage*).

UVM je využívána pro porovnání, že procesor popsaný v RTL se chová ekvivalentně jako jeho softwarový simulátor. Slouží jako dynamická knihovna připojená k simulaci. Používá se termín reference, častěji je ale označován jako *„golden model“* nebo *„golden reference“*. Tato reference slouží jako vzorový model pro RTL reprezentaci digitálního obvodu. Názorně ukázáno na obrázku 4.1.



Obrázek 4.1: Generované UVM verifikační prostředí ve firmě Codasip

Verifikace se provádí vždy nad konkrétním programem. Typicky se verifikace spouští nad mnoha různými programy – používají se fixní programy i náhodně generované posloupnosti instrukcí.

Testy UVM jsou prováděné v softwarové simulaci v simulátorech podporujících jazyk SystemVerilog. Provádět tyto testy s využitím prototypu FPGA není možné z těchto důvodů:

- Úzká vazba UVM na simulaci, kde UVM knihovny nejsou syntetizovatelné do podoby hardwaru, protože využívají konstrukce jazyka SystemVerilog, které nejsou syntetizované.
- Je nutností přistupovat k vnitřním komponentám procesoru. Už dříve zmíněné white-box testování, které není možné v prototypování pomocí FPGA.

4.2.2 RISC-V Debug testy

Jedná se o sadu testů vyvinutých ve firmě Codasip, která má za cíl ověřit funkčnost on-chip debug modulu v procesorech RISC-V. Ověřuje funkčnost on-chip debug modulu proti specifikaci, kterou je RISC-V Debug Specification[3]. Princip testů je takový, že v rámci testů se provádí akce s procesorem prostřednictvím on-chip debug modulu a kontrolují očekávané výstupy. Vše je zařízeno formou přístupu do speciálních registrů uvnitř RISC-V debug modulu a tento přístup se provádí přes rozhraní JTAG.

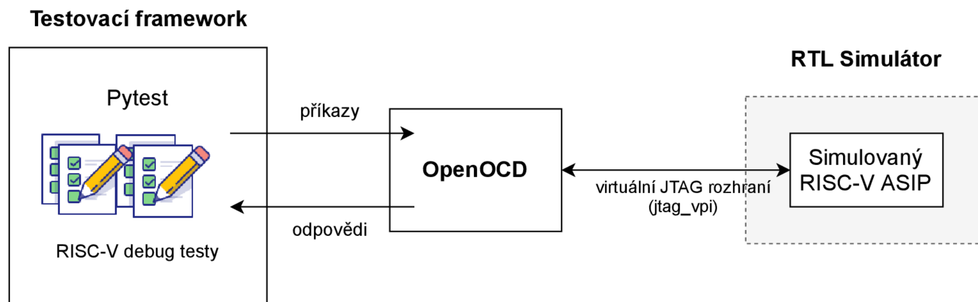
[3]https://github.com/riscv/riscv-debug-spec

Příklad jednoduchého testu RISC-V debug. Ověření funkčnosti watchpointu:

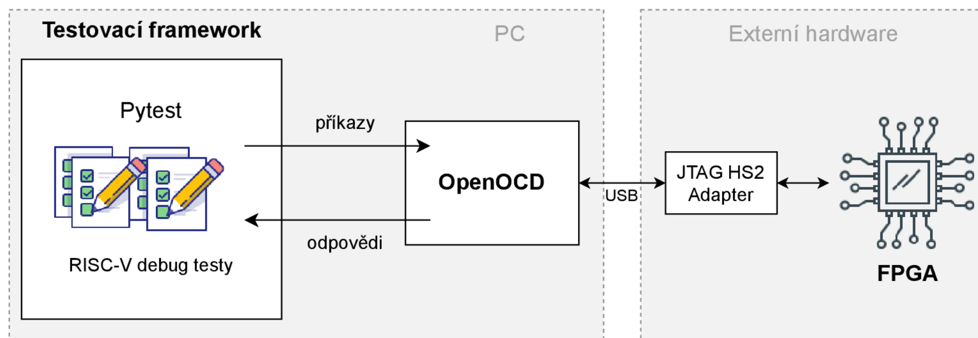
```
def test_watchpoint_write():
    reset_and_halt_cpu()
    load_program(watchpoint_test_program)

    # program, ktery zapisuje na adresu v~pameti 0x3000
    place_watchpoint(address=0x3000, operation=write)

    resume_cpu()
    sleep_ms(500)

    assert cpu_is_halted()
    assert cpu_halt_reason() == HALT_REASON_WATCHPOINT
```

Výpis 4.1: Pseudokód vybraného testu ze sady RISC-V debug testů

Jako software pro komunikaci s on-chip debug modulem procesoru slouží OpenOCD. Jednotlivé testy jsou tvořeny sekvencemi povelů pro OpenOCD, přičemž OpenOCD přijímá povely v jazyce TCL. Testy jsou napsány v jazyce Python a spouštějí se prostřednictvím knihovny Pytest. V současnosti jsou testy RISC-V debug spouštěny s využitím RTL simulace procesoru. OpenOCD s RTL simulací je propojeno prostřednictvím „virtuálního" JTAG protokolu. Jedná se o jednoduchý protokol `jtag_vpi` sloužící pro přenos operací rozhraní JTAG pomocí protokolu TCP. Názorně zobrazeno na obrázku 4.2.



Obrázek 4.2: Aktuální řešení testování s RISC-V debug testy



Obrázek 4.3: Rozšířená varianta RISC-V debug testů o využití prototypu pomocí FPGA

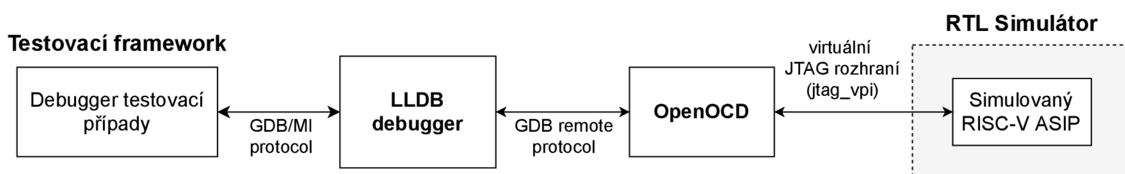
Po přidání prototypování v FPGA do současného řešení bude systém rozšířen o zapojené FPGA jak je vidět na obrázku 4.3. Testy RISC-V debug jsou vhodným kandidátem na spouštění prostřednictvím hardware prototypu z důvodů:

- Jedná se o testování typu black-box, není zde nutnost zkoumat vnitřní signály procesoru.
- Interakce s procesorem probíhá prostřednictvím JTAG rozhraní a on-chip debug modulu a toto rozhraní je u FPGA prototypu dostupné.

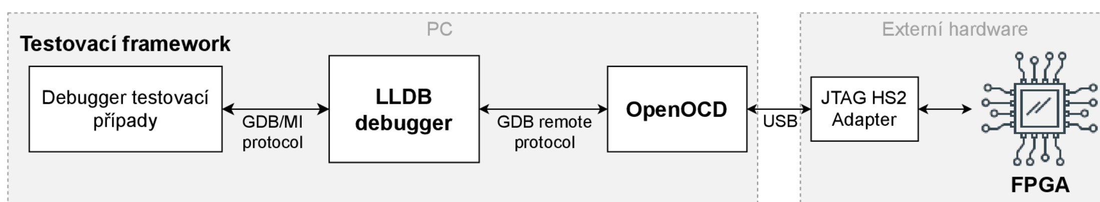
4.2.3 Debugger testy

Debugger testy jsou sada testů vyvinutá ve firmě Codasip, která ověřuje spolupráci více hardware a software komponent nutných při ladění programů běžících na vestavěných procesorech. Jde tedy o integrační testy, které se sestávají z operací na vyšší úrovni abstrakce než testy RISC-V debug, popsané v sekci 4.2.2. Debugger testy rozšiřují RISC-V debug testy tak, že je přidána další komponenta debugger (LLDB). Debugger je software využívaný pro vyhledávání chyb v kontrolovaném prostředí. LLDB je debugger vyvinutý v rámci projektu LLVM (alternativa k GDB[4]).

Debugger testy jsou tvořeny sadou povelů pro debugger LLDB, pro komunikaci s ním se využívá protokol `GDB/MI`[5] (MI je angl. zk. *Machine Interface*). LLDB potom komunikuje s OpenOCD prostřednictvím protokolu *GDB remote*[6]. OpenOCD pro LLDB zprostředkovává přístup k funkcím on-chip debug modulu v hardwaru. Stejně jako v předchozím případě jsou testy nyní spouštěny s využitím RTL simulace procesoru. OpenOCD je připojeno do simulace prostřednictvím protokolu `tag_vpi`, jak je vidět na obrázku 4.4.



Obrázek 4.4: Komunikace mezi potřebnými komponenty k debugger testům v aktuální stavu



Obrázek 4.5: Debugger testy rozšířené o možnost běhu nad prototypem procesoru v FPGA

Debugger testy je možné spouštět na hardware prototypu s využitím FPGA, platí zde stejné důvody jako u RISC-V debug testů. Situaci po nahrazení RTL simulace za FPGA prototyp je znázorněna na obrázku 4.5. Po této úpravě se nezmění samotné testy (sekvence povelů pro LLDB), konfigurace LLDB ani komunikace LLDB s OpenOCD. Jinými slovy

[4] https://www.gnu.org/software/gdb/documentation/
[5] https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html
[6] https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html#Remote-Protocol

pro LLDB je zcela transparentní, zdali pracuje s fyzickým nebo simulovaným hardwarem. Jediná změna je nutná v konfiguraci OpenOCD, kdy komunikační protokol `jtag_vpi` je nahrazen prací s hardwarovým adaptérem JTAG-HS2, připojeným do USB portu počítače.

4.3 Důvody pro implementaci systému

V současné době již ve firmě Codasip existuje infrastruktura pro automatizované spouštění testů a tyto testy běží na virtuálních strojích nebo v docker kontejnerech[7]. Je potřeba do aktuálního firemního testovacího systému zakomponovat fyzický hardware (vývojové desky s čipy FPGA). Tento fyzický hardware totiž není možné přímo zapojit do aktuálního systému, protože je veden z větší části virtuálními stroji, popřípadě kontejnery. Z tohoto důvodu je firmou Codasip požadováno, aby vznikl centrální systém umožňující spravovat více FPGA desek, ke kterým bude možné vzdáleně přistupovat a vykonávat na nich automaticky spouštěné testy.

Vývojáři chtějí mít možnost připojit se vzdáleně k jednotlivým deskám FPGA, spravovat tyto zařízení, mít přehled o jejich aktuálním stavu a možnost je nakonfigurovat. Mimo správu zařízení je další stěžejní funkcí spouštět testy na připojených deskách FPGA. Některé testy není možné provádět v simulátoru, ale v FPGA lze, jak už bylo zmíněno v kapitole 3.

Je zde také možnost převodu z rozhraní JTAG na rozhraní Ethernet. Tento způsob se firmě Codasip ale nehodí, protože nelze spravovat více desek najednou a nelze spravovat frontu čekajících úloh ke zpracovaní.

Motivace zavedení metody prototypování v FPGA

- Zmenšit náklady za nutné licence pro simulační software (RTL simulátory).
- Zrychlení běhů testů. To se však nemusí projevit u všech testů, protože rychlost testů není dána jen výkonností hardwaru, ale také rychlostí, jakou jsou softwarové nástroje (tj. testovací framework, debugger LLDB a nástroj OpenOCD) s hardware schopny interagovat.
- Testování s využitím fyzických debug ataptérů, které není možné připojit k simulaci (například v této práci využívaný JTAG-HS2)
- Je vhodné mít otestovaný procesor jako hardware, nejen v simulátoru, neboť zákazníci firmy Codasip často s procesory pracují právě ve formě FPGA prototypů.

4.4 Existující řešení

Požadavky na systém vycházejí ze specifických potřeb firmy Codasip a z konkrétních typů testů. Proto na trhu není k dispozici hotové řešení, které by bylo možné přímo použít. Je však možné nalézt produkty, které mají některé podobné rysy se systémem vyvíjeném v této práci, např. produkt Aldec HES-DVM.

[7]Docker je produkt sloužící pro virtualizaci na úrovni operačních systémů. Pomocí balíčků (kontejnerů) umožňuje zprostředkovat software pro různé operační systémy v definovaném prostředí kontejneru https://www.docker.com/

Aldec HES-DVM

HES-DVM [1] je komerční produkt kombinující softwarové simulační nástroje a hardwarové FPGA desky, které slouží k prototypování hardwarových návrhu a k akceleraci softwarové simulace (HW emulace). Poskytuje moderní emulační standardy a výkonné technologie FPGA, aby vývojové týmy pro návrh hardwaru a softwaru měly včasný přístup k hardwarovému prototypu návrhu. Souběžně mohou spolupracovat na vývoji a ověřování kódu na vysoké úrovni s přesností RTL a rychlostně efektivní emulací SoC nebo prototypovými modely, což zkracuje dobu testování a snižuje riziko nezachycené vady v návrhu. Produkt poskytuje verifikačním týmům více režimů použití, včetně technik emulace a fyzického prototypování, což umožňuje SoC týmům pracovat na jedné platformě.

Obrázek 4.6: Přehled využití HES-DVM. Zdroj [1].

Podle údajů výrobce nabízí produkt HES-DVM tuto funkčnost:

- Podpora pro akcelerační desky Aldec HES[8], ale i desky třetích stran nebo vlastně vyrobené.
- Hardwarová akcelerace pro simulaci (simulátory od firmy Aldec i simulátory třetích stran).
- Podpora jazyků SystemVerilog, Verilog a VHDL.
- Funkce ladění.
- Rychlé nastavení pro prototypování více FPGA.
- Monitorování využitých logických zdrojů a propojení.
- Uživatelská rozhraní pro skriptování GUI a TCL.

Umožnění škálovatelnosti je hlavním cílem vývojového týmu HES-DVM a díky tomu je toto řešení jedinečné. Aldec není omezen na pevnou vyhrazenou platformu pro emulaci hardwaru, pokračuje ve vývoji otevřené architektury, kterou lze rychle přenést na novou

[8] https://www.aldec.com/en/products/emulation/hes_fpga_boards

generaci technologie FPGA a také použít s vlastními prototypovými deskami vyrobenými na zakázku.

Aldec HES-DVM nabízí velice robustní řešení, které je ale také řádně zpoplatněno. HES-DVM není prodávaný jen v jedné verzi, ale svoji funkčnost nabízí po odstupňovaných balících podle funkcionalit uvnitř. Nabízí 4 různé balíky, od prvního, kde je obsažena funkcionalita, až po vyšší verze, které přidávají další funkce. Jedná se o balíčky Prototyping, Acceleration, Emulation a Elite. Codasip projevil zájem o funkcionalitu balíku Emulation, přičemž cenová nabídka byla v řádech vyšších desítek tisíc dolarů až přesahujících stovky tisíc dolarů.

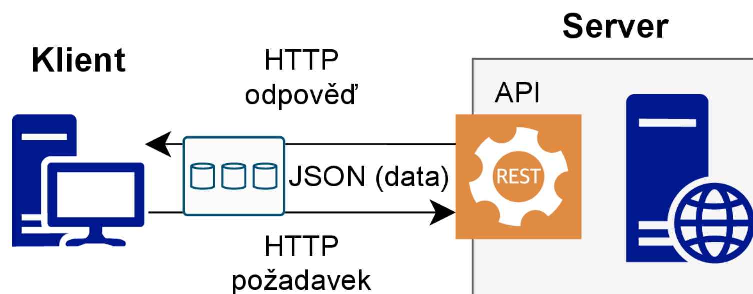
Z toho vyplývá, že u tohoto produktu by nebylo dosaženo jednoho z významných cílů, kterým je zmenšit výdaje vynaložené za licence softwarových simulátorů, ale přibyla by licence za jiný software. Při bližším pohledu na toto řešení by bylo problematické zapojit ho do aktuálního firemního systému na testování a některé specifické funkce jsou zde navíc a některé naopak chybí.

Kapitola 5

Návrh testovacího systému pro farmu FPGA

5.1 Architektura systému

Systém byl navržen tak, aby uživatel systému mohl vzdáleně ovládat FPGA desky, které budou připojené na jednom místě a bude možné k nim přistupovat v reálném čase. K tomu byla vybrána architektura klient-server (viz. Obrázek 5.1), která zde komunikuje přes protokol *HTTP* (angl. zk. *Hypertext Transfer Protocol*). Aplikační rozhraní *API* (angl. zk. *Application Programming Interface* serveru, dále jen API, na které klient posílá požadavky HTTP. Tato technologie používá podmnožinu protokolu HTTP. Klient zprostředkovaně přes aplikaci implementovanou v systému používá pro vstup příkazovou řádku, pomocí které může posílat na API serveru zmíněné požadavky s využitím protokolu HTTP. Data posílaná od klienta na server jsou ve formátu *JSON* (angl. zk. *Javascript Object Notation*), který tyto data serializuje [9], a proto je vhodný pro jejich přesun na zmiňovaném protokolu.

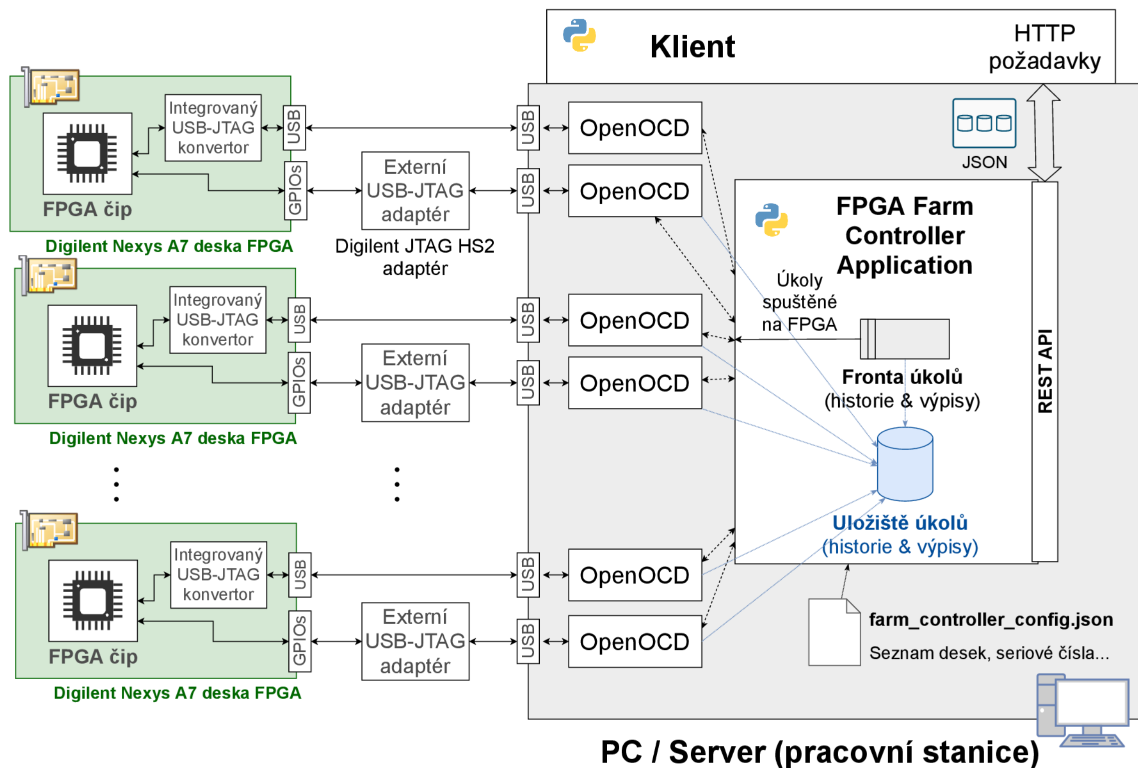


Obrázek 5.1: Architektura systému

Použití HTTP protokolu svádí k využití webových stránek, tento systém však primárně slouží pro rozšíření testování možností testovat v FPGA, proto nejsou webové stránky vhodné. Aktuálně jsou veškeré testy implementované v jazyce Python a jsou spouštěny testovacím frameworkem vyvinutým ve firmě Codasip. Tyto testy musí být upraveny tak, aby byly schopné dotazovat se na běžící implementovaný server. Je tedy nutné, aby systém bylo možné ovládat jednoduchými příkazy prostřednictvím klientské části a odeslaná a přijatá data byla v serializovaném formátu, v tomto případě dříve zmíněný formát JSON. Existují i další technologie na serializaci dat například: XML, YAML a další [9, 17].

5.2 Způsob využití systému

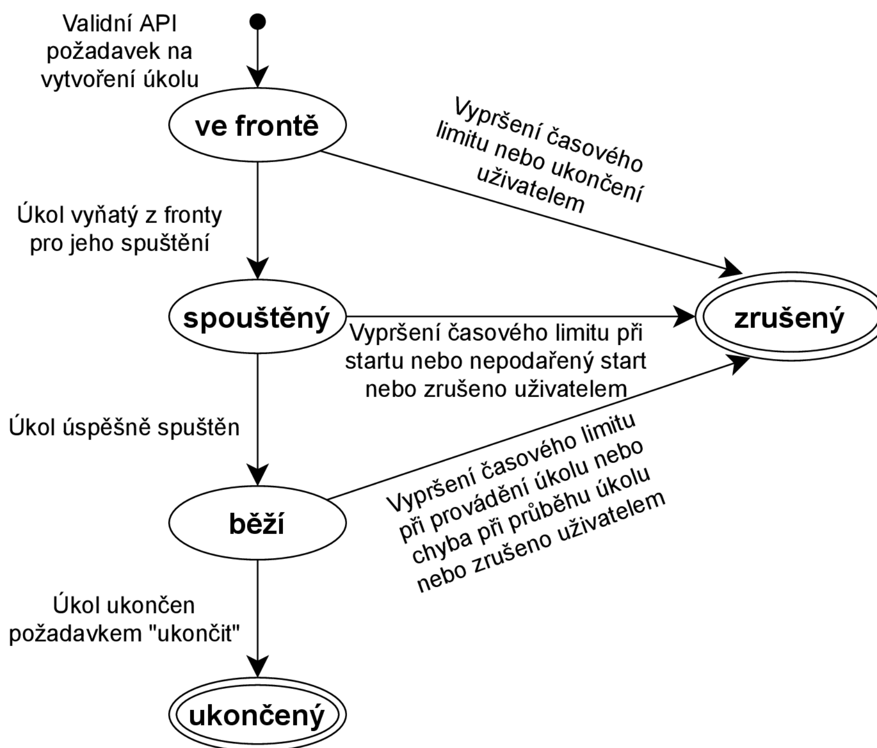
Serverová část systému je spuštěna na některém ze serverů firmy Codasip nebo ve vývojové fázi na osobním počítači, který je zpřístupněný na lokální síti. V obou případech jsou k těmto zařízením připojeny přes USB desky FPGA. V případě osobního využití pro správu FPGA desek si uživatel u sebe nainstaluje klientskou aplikaci a přes příkazovou řádku zadává podporované příkazy. Jako ověření úspěšných spuštění dostane uživatel od serveru odpověď, z které je zřejmé, jak zadaná operace dopadla. V dalším případě jsou požadavky na server odeslané automaticky s pomocí testovacího frameworku, který firma Codasip využívá. Pro konkrétní ukázku komunikace systému a jeho podčástí slouží obrázek 5.2, kde je ukázána nejen dříve popsaná komunikace klienta a serveru. Konkrétním požadavkům, které API podporuje, je věnovaná sekce 5.3.



Obrázek 5.2: Schéma implementovaného systému pro testování na FPGA

Hlavním z požadavků na server je vytvoření úkolu. Takový úkol může znamenat nahrání souboru bitstream do desky a následné spuštění testů nebo aplikace v FPGA, kterou uživatel s požadavkem pošle na server. Požadavek na vytvoření úkolu server vyhodnotí tak, že daný úkol zařadí do fronty a zároveň uloží potřebná data do databáze, jejíž schéma je na obrázku 5.4. Úkoly mají 5 stavů popsaných na obrázku 5.3, kterých mohou nabývat. Při přijetí validního požadavku systém změní stav úkolu na „ukládání“, kde jsou data ukládána do databáze. Stav slouží pro nenarušení konzistence dat ostatními procesy, které běží současně s obsluhou úkolů. Po přidání dat do databáze je stav úkolu změněn na „zařazen do fronty“. Zde je vhodné zmínit, že na serveru je opakovaně spouštěný proces, který kontroluje volné desky a v případě, že volné jsou, je úkolu zařazeném ve frontě přiřazena některá z nich. Je zde i možnost vyvolat tento proces ručně. Při přebrání úkolu z fronty dostane úkol status „startuje“. Po přiřazení desce změní úkol stav na „běží“. Po úspěšném ukončení nebo

ukončení vyvolaném uživatelem, se úkol dostane do stavu „ukončen“. Ze všech zmíněných stavů kromě stavu „ukončen“ je možné dostat úkol do stavu „přerušen“. Jednotlivé situace jsou popsané na obrázku 5.3.



Obrázek 5.3: Možné stavy úkolů a přechody mezi nimi

Ve chvíli, kdy si úkol zabere desku a je ve stavu „běží“, tak s pomocí OpenOCD popsaného v sekci 4.2.2, se naváže spojení s deskou FPGA a nakonfiguruje se předaným souborem bitstream. OpenOCD zde figuruje jako software umožňující komunikaci pro konfiguraci FPGA čipu přes integrovaný USB-JTAG konvertor na desce a také jako software pro komunikaci s čipem FPGA. Komunikace s čipem se provádí až po jeho konfiguraci. Po navázání komunikace pro ladění čipu OpenOCD otevře komunikaci na sebou zvolených portech a umožní tím příchozí komunikaci s aplikacemi mimo systém. Aby bylo OpenOCD schopno komunikace s čipem FPGA, musí zde být externí USB-JTAG adaptér, který přijímaná data z desky konvertuje, aby byla čitelná pro rozhraní USB. Naopak odesílaná na desku konvertuje tak, aby jim rozumělo rozhraní JTAG pro On-chip debugging. V systému je využit adaptér JTAG-HS2 zmíněný v sekci 2.6. Pro oba případy je využito OpenOCD z důvodu vývoje na operačním systému Windows. OpenOCD může zastávat tyto funkce díky jeho konfigurovatelnosti. Jednou z alternativ pro konfiguraci zde využitých desek je software *Digielent Adept* od firmy Digilent, který bude v dalších fázích vývoje také využíván.

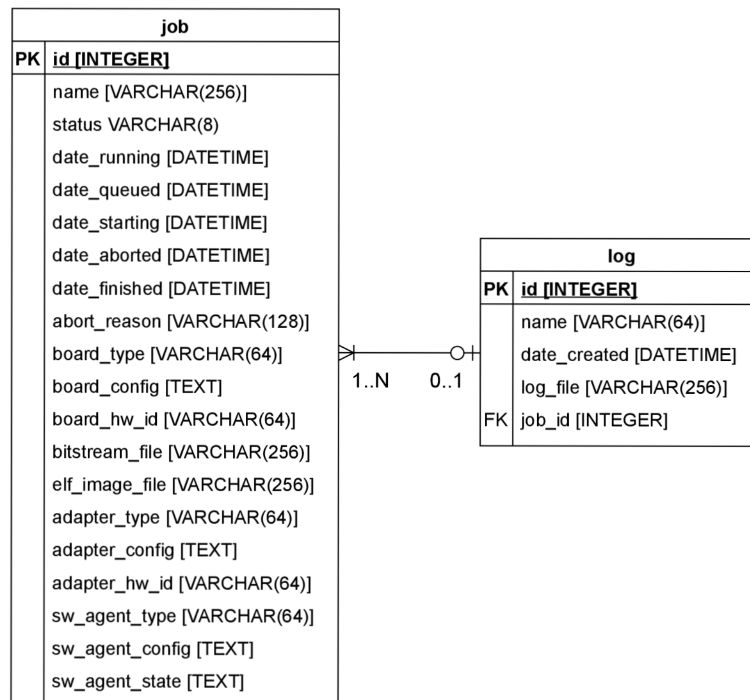
5.3 Aplikační rozhraní

Pro API byla zvolena technologie *RESTful* API (častěji používané *REST*) [28], která je založená na sadě architektonických omezení REST (angl. *Representational state transfer*). Když je požadavek klienta vytvořen prostřednictvím REST API, přenáší data na koncový

bod (angl. *endpoint*) v jednom z několika formátů prostřednictvím protokolu HTTP, jedná se o formáty: JSON, HTML, XLT, Python, PHP nebo čistý text. API systému slouží na správu úkolů, které lze s pomocí klienta vytvářet a odpovědi ze serveru chodí ve formátu JSON. API přijímá požadavky typu `POST` a `GET`, které zastávají následující funkce:

- Vytvoření úkolu (POST): Pomocí tohoto požadavku je na serveru vytvořen úkol. Aby byl úkol vytvořen, potřebuje od klienta zadat data, která se pošlou v rámci HTTP požadavku v podobě JSON.
- Zobrazení úkolů a jejich stavů (GET): Jako odpověď klient obdrží úkoly a jejich stavy.
- Status úkolu (GET): S pomocí reference (identifikátor úkolu) klient získá status úkolu.
- Detail úkolu (GET): Zadáním identifikátoru klient získá všechny data o úkolu uchovávaná v databázi.
- Uzavřít úkol (POST): Zadáním identifikátoru klient dokončí běh úkolu.
- Zrušit úkol (POST): Zadáním identifikátoru klient zruší úkol.
- Získat výpis (POST): Zadáním identifikátoru klient obdrží výpisy (angl. *logs*) z průběhu vykonávání úkolu.

5.4 Databázové schéma



Obrázek 5.4: Schéma databáze systému

Databáze je velmi jednoduchá a obsahuje jen dvě tabulky a její schéma je na obrázku 5.4. První tabulka *job* obsahuje data, která uživatel zadal při vytváření úkolu a data vznikající

při jeho běhu, jedná se o časové momenty při přechodu mezi stavy úkolů a identifikátory desky a adaptéru. Bitstream a soubor `elf` reprezentující spouštěnou aplikaci v FPGA čipu jsou soubory, které uživatel pošle v požadavku na server. Soubory jsou před odesláním zakódovány do formátu `base64`. Na straně serveru jsou pak dekódovány a uloženy v jeho souborovém systému. V databázi je uložena jen cesta k souborům, protože nemá smysl v databázi ukládat tak velké soubory. V druhé tabulce *log* jsou zaznamenané výpisy z průběhu vykonávání úkolu a taktéž potřebné časové údaje. Výpisy jsou uloženy stejně jako soubory ve stejné složce a v databázi je jen cesta k výpisům.

Kapitola 6

Implementace

Aby bylo dosaženo celého procesu testování procesorů v FPGA, je nutné využít také aplikace a systémy existující mimo implementaci tohoto systému. Implementovaný systém se sestává z dvou základních částí. V této kapitole jsem se zaměřil právě na ně. Jedná se o klientskou část systému a část *FPGA Farm Controller Application* fungující jako server. Obrázek 5.2 zobrazuje, jak zapadají do celého systému. Mimo implementaci je už vytvořený testovací framework, který bude odesílat příkazy s využitím klienta na otevřené porty serveru, kde poběží serverová část aplikace.

Klient je aplikace implementovaná v jazyce Python, která zastává funkci příkazové řádky, kde při spuštění nainstalované klientské aplikace uživatel specifikuje příkaz s argumenty, volbami a zadá nutná dat a poté příkaz spustí. Po spuštění příkazu klient na své straně ověří data od uživatele a odešle je na server, kde jsou znovu validovány. Požadavky HTTP jsou vytvářeny s pomocí knihovny `requests`[1]. Obecně jsou funkce popsány v sekci 5.3.

Server je aplikace, která obsluhuje požadavky. Aplikace je vytvořena s pomocí webového frameworku `flask`[2], napsaného v jazyce Python. Jsou zde dvě skupiny koncových bodů, kde každý koncový bod odpovídá jednomu z požadavků od klienta. Jedna skupina vybírá požadovaná data z databáze a posílá je zpět klientovi. Druhá skupina obsluhuje úkoly, které server na popud klienta vytvoří. Druhá skupina tvoří hlavní funkčnost systému. Nejdřív je potřeba vytvořit úkol a uložit ho do databáze, aby požadavky z první skupiny mohly úkol z databáze získat.

6.1 Klient

Klient je složen ze dvou částí. Jedna část je implementovaná s pomocí modulu `click`[3], který zastává funkci příkazové řádky *CLI* (angl. *Commandline Interface*). Umožnuje spouštět klienta s definovanými příkazy a jejich přepínači a argumenty, kterými definuje a omezuje data pro požadavek na server. Následující tabulka 6.1 spojuje příkazy z aplikace klienta s API serveru.

Druhá část je třída `Client` sloužící k odesílání požadavků využitím modulu `requests`. Soubor s implementací první části, kde je využit `click`, má být zpravidla co nejjednodušší, proto je potřebná funkcionalita implementována ve třídě `Client` a jejích metodách. Tato

[1] https://docs.python-requests.org/en/master/
[2] https://flask.palletsprojects.com/en/1.1.x/
[3] https://click.palletsprojects.com/en/7.x/

Příkaz	Odpovídající endpoint
`create-job`	`/jobs/new`
`close-job`	`/jobs/<int:job_id>/close`
`abort-job`	`/jobs/<int:job_id>/abort`
`job`	`/jobs/<int:job_id>/details`
`status`	`/jobs/<int:job_id>/status`
`list-jobs`	`/jobs`
`logs`	`/logs/<int:job_id>`

Tabulka 6.1: Přehled implementovaných koncových bodů serverového API (tabulka 6.2)

třída a její metody jsou pak volány v první části klienta. Třída data zpracovává a validuje. V sekci 5.3 se zmiňuji o zakódování předaných souborů (bitstream a `elf`) do kódování `base64`. To je umožněno pomocí stejnojmenného vestavěného modulu. Následně se soubory musí převést z typu `byte` na typ `string`, aby je bylo možné odeslat ve formátu JSON v požadavku protokolu HTTP. S příkazy a daty přijatými od uživatele klient odešle požadavky na server s využitím modulu `requests`.

Klient definuje 7 základních příkazů a to: `create-job`, `list-jobs`, `job`, `status`, `close-job`, `abort-job` a `logs`. Podrobnosti o jejich možnostech spuštění lze zjistit s pomocí volby `--help`.

6.2 Server

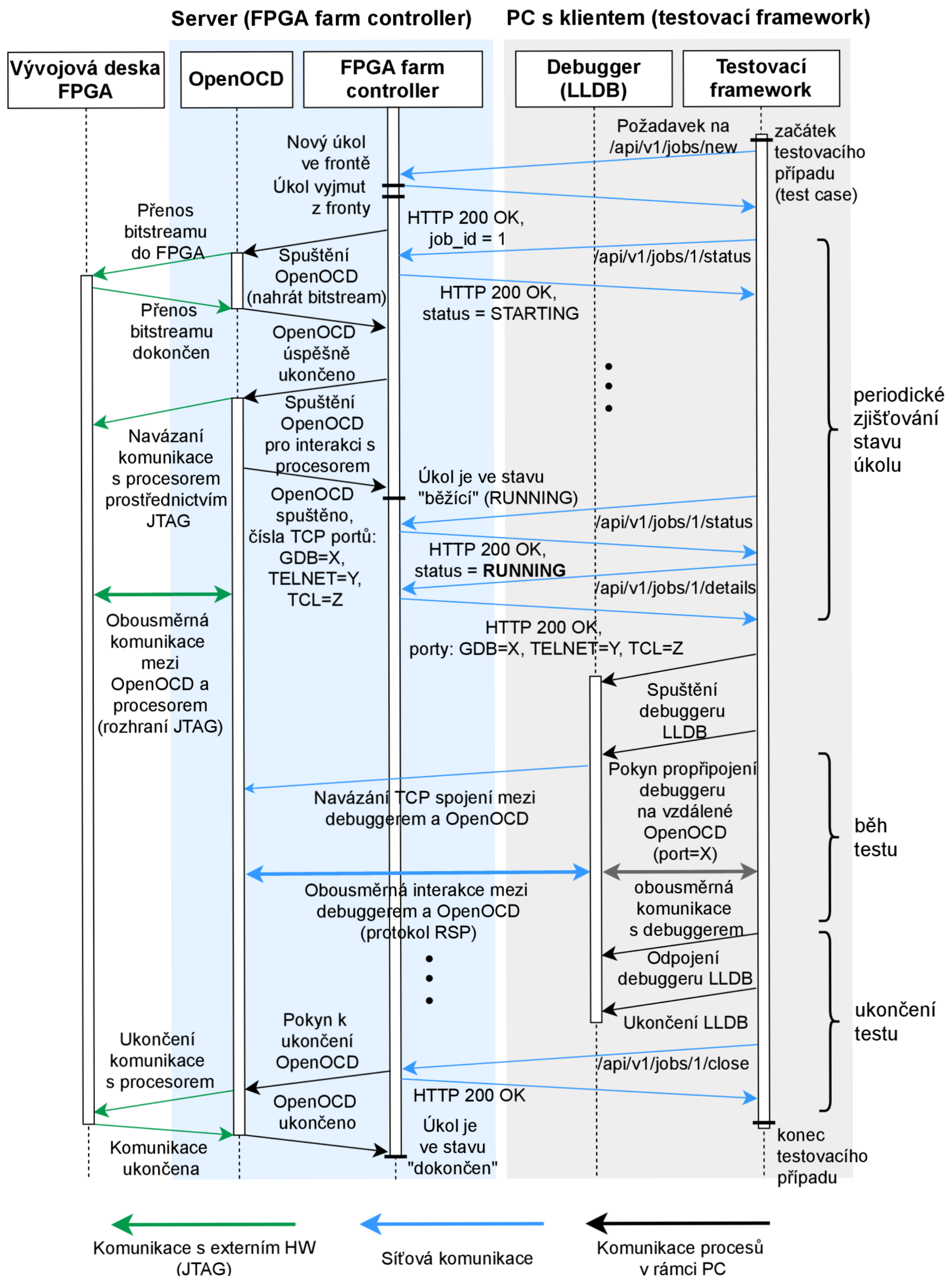
Serverová aplikace je implementovaná pomocí webového frameworku `flask`, který umožňuje jednoduše tvořit API a jeho koncové body. Pro koncové body se definuje, jaké typy požadavků HTTP přijímají. V systému jsou zastoupeny jen typy `POST` a `GET`. Před definováním API je potřeba vytvořit instanci třídy, která modeluje aplikaci (server) připravenou webovým frameworkem. Vytvářenou instanci aplikace lze nakonfigurovat a je možné nastavit například: adresu databázového serveru, testovací mód serveru, název serveru, bezpečnostní klíč. Na instanci se pak navážou dříve zmíněné koncové body a tímto způsobem se definuje API serveru.

Typ požadavku	Endpoint	Popis
`POST`	`/jobs/new`	Vytvoření nového úkolu
`POST`	`/jobs/<int:job_id>/close`	Dokončení úkolu
`POST`	`/jobs/<int:job_id>/abort`	Zrušení úkolu
`GET`	`/jobs/<int:job_id>/details`	Získání podrobností úkolu
`GET`	`/jobs/<int:job_id>/status`	Získání statusu úkolu
`GET`	`/jobs`	Získání seznamu úkolů
`GET`	`/logs/<int:job_id>`	Získání výpisu z úkolu

Tabulka 6.2: Přehled implemntovaných koncových bodů serverového API

V metodě, která vytváří instanci aplikace, je také spuštěn periodicky se opakující proces. Tento proces je implementovaný s pomocí modulu `threading`[4], který umožňuje správu procesů (zámky, semafory) a ovládat procesy ve více vláknech. Proces slouží k vybírání úkolů z fronty tak, že se deska přiřadí úkolu a tím změní stav úkolu na „spouštění“.

[4] https://docs.python.org/3/library/threading.html



Obrázek 6.1: Schéma toku procesů v systému

Dále jsem se zaměřil na vytváření úkolu a jeho zpracování serverem a celý proces je zobrazen na obrázku 6.1. Po přijetí požadavku aplikace vloží úkol do fronty, která je tvořena seznamem úkolů vybraných z databáze a seřazena podle času jeho vytvoření, aby se jednalo o metodu první dovnitř, první ven *FIFO* (angl. zk. *First In First Out*) reprezentující frontu. Zmíněný proces se opakovaně dotazuje, zda nečeká úkol ve frontě a případně ho z ní vyjme. Tento proces může být vyvolán i na zavolání tzn. uživatelským požadavkem. Ve chvíli, kdy je úkol vybrán z fronty, server kontroluje přístupné desky FPGA, které by úkol obsloužily. Dostupné desky jsou definované v konfiguračním souboru ve formátu JSON, z kterého si server bere potřebná data. Dokud server nenajde volnou desku, úkol čeká ve frontě. Volnost desky je definovaná tím, že k sobě nemá přiřazené žádné identifikační číslo úkolu.

Po tom, co je úkolu přiřazena deska, OpenOCD je nakonfigurováno serverem vygenerovaným konfiguračním souborem, který je vyplněn konfigurací od klienta. Tento konfigurační soubor zajišťuje správnou funkcionalitu OpenOCD, v tomto případě pro konfiguraci čipu FPGA. Na generování souboru slouží soubor `config_generator.py`, o kterém se zmiňuji v podsekci 6.2.2. Následně je OpenOCD spuštěno jako podproces s využitím modulu `subprocess` a nakonfiguruje FPGA čip souborem bitstream. Dále je vygenerován další konfigurační soubor pro OpenOCD, který ho nastaví na mód On-Chip debugging a spustí ho jako podproces, který bude očekávat příkazy od uživatele pro ladění na čipu FPGA. Toto spuštění OpenOCD pro ladění není jednoduché. OpenOCD je v tomto případě spuštěno na pozadí jako tzv. démon (angl. *deamon*) a v tomto případě není jednoduché vyčítat jeho výpisy na standardní výstupy. Pro tuto funkcionalitu jsem využil kód v souboru `popen_wrapper.py`, který už je využit v jiných firemních systémech, tudíž není vytvořený mnou. O něm se zmíňuji v podsekci 6.2.3.

Hlavním předmetem celého tohoto systému je právě získat data z OpenOCD, přesněji porty, které OpenOCD na serveru otevře a přes ně umožní komunikaci s čipem FPGA. Ty lze vyčíst právě z jeho výpisů a po jejich vyčtení jsou uloženy v konfiguraci agenta (v tomto případě OpenOCD). Aby je klient získal, použije příkaz *job* a zadá identifikační číslo úkolu, což klientovi poskytne nejen porty, ale i všechny data, která s úkolem souvisí.

Ukončení ladění je možné vyvolat pomocí požadavku `close-job` a tímto se podproces ukončí a úkol bude převeden do stavu „dokončen“, následně se uloží do databáze. Všechny výpisy z ladění budou uloženy v pracovní složce serveru a budou přístupné přes požadavky z klientské aplikace.

6.2.1 Databáze

Na databázi je v systému využit jednoduchý relační databázový systém SQLite[5]. Databázový systém je obsluhována s pomocí sady *SQLAlchemy*[6] a s pomocí ní je využit objektově relační mapovač *ORM* (angl. *Object-Relational Mapper*). SQLAlchemy slouží jako sada nástrojů pro správu databáze. ORM přístup umožňuje jednat0 s tabulkami jako s objekty a přidává tedy možnost dědičnosti a dalších vlastností, které samotné relační databáze v SQL nemají. Díky využití SQLAlchemy, které podporuje mnoho databázových systémů, bude v budoucnu jednoduché přejít na jiný databázový systém. Protože to, na jakém systému funguje databáze, je v logice operací kódu odstíněno.

[5] https://sqlite.org/index.html
[6] https://www.sqlalchemy.org/

6.2.2 Generátor konfiguračních souborů pro OpenOCD

Generátor v systému slouží k vytvoření konfiguračních souborů pro program OpecOCD. Tyto soubory se musí generovat, protože je do nich potřeba dodat data, například identifikační číslo hardwaru či příkazy pro specifikaci chování OpenOCD. Sestavení souboru stojí na třídě `Template` z modulu `string`, který je součástí standardních Python knihoven. Tento šablonovací systém je jednoduchý a nabízí určit si vlastní oddělovač, kterým nachází v zadaném textu položky (proměnné) sloužící k nahrazení zvoleným řetězcem ze vstupu. Vlastní oddělovač je vhodný, protože konfigurační soubor pro OpenOCD uvnitř může obsahovat standardní znak dolaru $ pro proměnnou a další.

6.2.3 OpenOCD jakožto démon

Tato část je převzatá z firemních zdrojů, není tak implementovaná mnou. Po nakonfigurování čipu FPGA, je potřeba nastavit OpenOCD pro práci s on-chip debug modulem zmíněným v sekci 2.1.5. Problém spočívá v tom, že se OpenOCD pustí na pozadí v systému ve vlastním vlákně a je potřeba v čase pořád naslouchat na jeho standardní výstupy. Tuto funkcionalitu umožňuje třída `Popen` z modulu `subprocess`. Při spuštění procesu s pomocí `Popen` se vytvoří dvě vlákna, která monitorují chybový standardní a standardní výstup. Využití vláken je jediný spolehlivý přístup mezi různými platformami pro monitorování více procesů spuštěných paralelně. Pro sbírání výstupů je zde využita třída `Queue` z modulu `queue`, kam se výstupy ukládají a následně se z nich data vyčítají do vyrovnávací paměti nazývané *buffer*.

Kapitola 7

Testování systému

Testování systému bylo provedeno nejen kvůli ověření funkčnosti celého systému, ale také z důvodu porovnání dob trvání testů v FPGA a v simulátoru. Před započetím testování bylo předpokladem, že proběhne zkrácení doby testů. V případě testů RISC-V debug proběhlo razantní časové zlepšení oproti testování v simulátoru. U debbuger testů se doba testování v FPGA prodloužila. Testování proběhlo na vývojové desce FPGA Nexys A7-100T od výrobce Digilent a laptop využitý pro spouštění testů byl vybavený procesorem *Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz* a využíval 1 procesorové vlákno.

7.1 Manuální testování

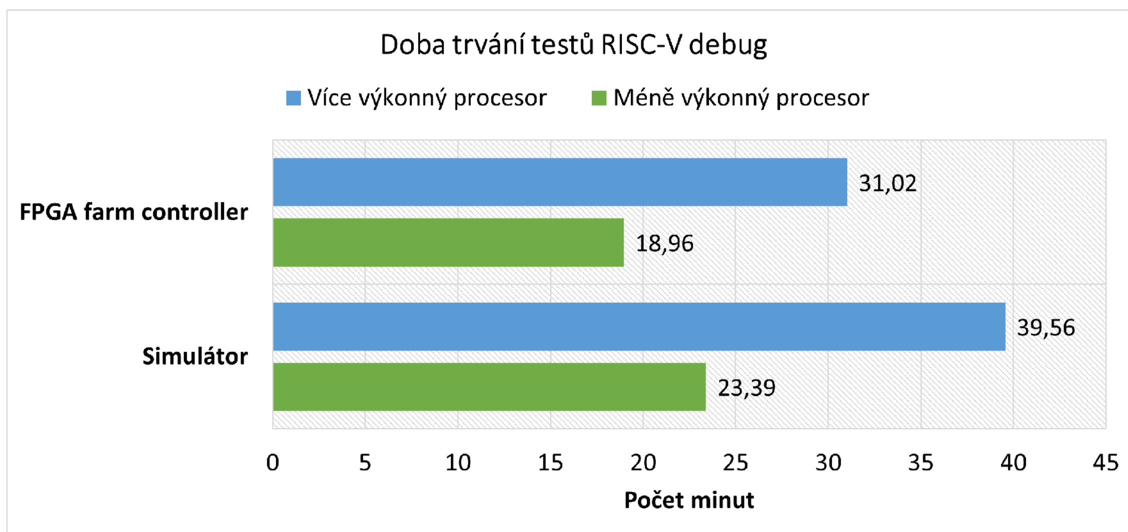
Manuální testování systému probíhalo tak, že klientem byly odesílány požadavky na server, následně byly validovány odpovědi ze serveru a zkontrolována jejich odpovídající výpovědní hodnota. Server byl při testování spuštěný na osobním počítači a klient z počítače ve stejné síti odesílal požadavky směrované na adresu serveru. Jako testovací případy použití byly využity všechny varianty požadavků, které otestovaly celé rozhraní API v různých časových okamžicích stavu serveru. Jednotkové testy pro tento případ testování nejsou v současné době implementované, ale systém byl navrhnut a z velké části implementován tak, aby jeho části byly testovatelné jednotkovým způsobem.

7.2 Systém integrovaný v testovacím frameworku

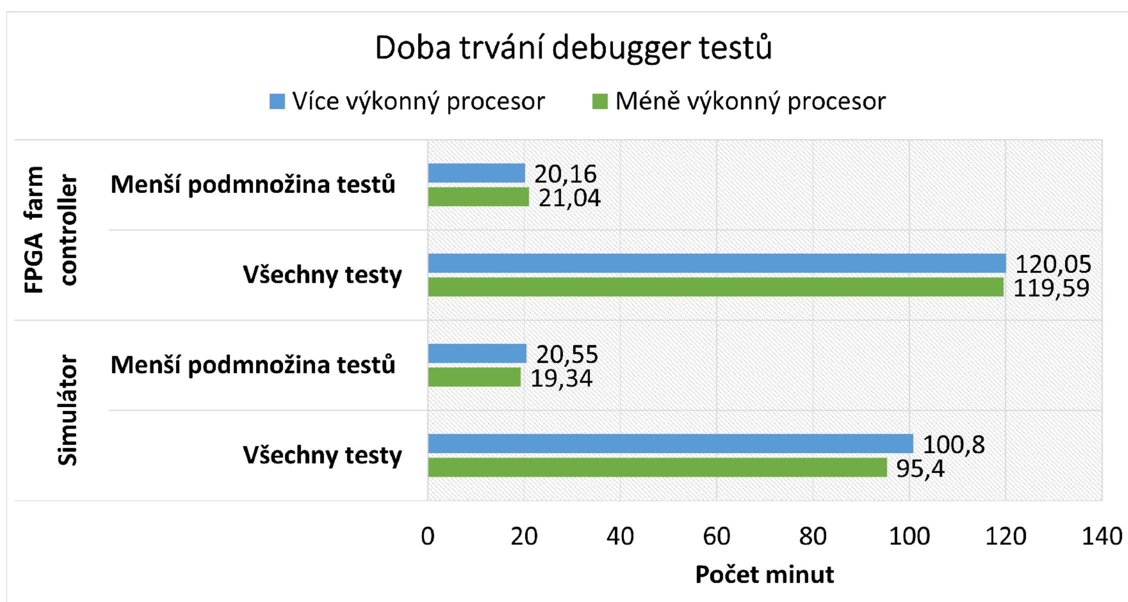
Systém byl také vyzkoušen s pomocí dříve zmíněného testovacího frameworku firmy Codasip. Toto testování zároveň ověřilo, že systém odpovídá požadavkům zjištěným při analýze a je možné ho využít ve firemním prostředí. V rámci tohoto testování byly využity existující testy, které musely být upraveny pro využití implementovaného systému. Úprava testů nebyla provedena mnou. Jedná se o testy RISC-V debug vysvětlené v sekci 4.2.2 a debugger testy vysvětlené v sekci 4.2.3. Spouštění testů posloužilo jako forma integračních testů.

Doba provádění těchto testů s pomocí implementovaného systému využívající desky FPGA byla porovnána s dobou trvání těchto testů v simulátoru. Očekávaný výstup byl takový, že doba testů se měla zkrátit. V případě testů RISC-V debug se jednalo o relativně velké zrychlení, jak jde vidět na grafu 7.1. Příčinou zrychlení je, že při použití desek FPGA není nutné v počítači provádět start simulátorů a jejich obsluhu, která je náročná na paměť a výkon. Dalším důvodem je využití plného paralelismu v hardware. Nevýhodou při testování na FPGA je konfigurace čipu – nahrávání souboru bitstream. Každý z testovacích případů

testů debuggeru vyžaduje znovu nakonfigurovat čip pomocí souboru bitstream, to způsobuje vyšší režii věnovanou jeho nahrávání. Z měření hodnot bylo zjištěno, že tato režie navíc razantně prodlužuje dobu běhu testů. Aby se doba zkrátila, bude potřeba přepsání debugger testů. Další z optimalizací může být zkrácení periody procesu pro kontrolu volných úkolů. Naměřené hodnoty z testů lze vyčíst z obrázku 7.2.



Obrázek 7.1: Porovnání doby běhu testů RISC-V debug v simulátoru a v FPGA



Obrázek 7.2: Porovnání doby běhu debugger testů v simulátoru a v FPGA

7.3 Budoucí vylepšení systému

Budoucím vylepšením systému by mohlo být využití externích aplikací, které zastávají funkcionality v současnosti naimplementové s pomocí knihoven jazyka Python. Jedná se

například o nahrazení správce front zprostředkovatelem zpráv (angl. *message broker*). Jedním takovým je například `RabbitMQ`[1], který pracuje asynchronně a zařizuje komunikaci komponent v systému. Nadále je možné využít jiného poskytovatele serveru. V implementovaném systému je teď poskytován frameworkem `flask`. Pak je zde část systému, která se stará o správu jednotlivých úkolů a přeměnu jejich stavů v průběhu jejich vykonávání. Těmito vylepšeními by se serverová část aplikace rozdělila na více částí, jež jsou samy o sobě dobře uchopitelné a testovatelné. Dalšími rozšířeními by pak mohly být:

- Automatické generování souboru bistream,
- plánování úkolů,
- izolace běhu OpenOCD od zbytku systému,
- přidání proiority úkolům,
- zabrání jedné desky více úkoly.

Dále bude vhodné mít zdroje rozdělené do jednotlivých týmu po expanzi systému v rámci firmy a po přikoupení dalších desek FPGA. To znamená personalizaci přístupu ke zdrojům pro jednotlivé uživatele. Při vyšším využití systému bude více dat, které bude systém uchovávat, bude tedy vhodné přejít na jiný databázový systém. Změna databázového systému nebude složitá operace z důvodů zmíněných v sekci 6.2.1. Pro ukládání dat jako jsou výpisy (*logs*), je vhodný například databázový systém `MongoDB`[2]. Aktuálně firma Codasip pro svoje potřeby testování využívá jen desku Nexys A7-100T od výrobce Digilent. Do budoucna se předpokládá využití i jiných desek, to znamená, že bude potřeba přidat podporu pro další typ desek. To stejné platí pro typ adaptérů JTAG. Příhodné bude také přidání podpory dalších sofwarových agentů, jako je OpenOCD. Může se jednat například o `J-Link GDB server`[3] a `JLink Commander`[4].

Firma Codasip aktuálně vyvíjí svůj testovací systém, který bude komplexní a bude možné do něj zapojit různorodé systémy. Do tohoto nového systému bude v budoucnu FPGA farm controller zapojen a využíván. Pro jeho zapojení do testovacího systému budou potřeba úpravy serverové aplikace i klienta.

[1] https://www.rabbitmq.com/
[2] https://www.mongodb.com/blog/post/mongodb-is-fantastic-for-logging
[3] https://www.segger.com/products/debug-probes/j-link/tools/j-link-gdb-server/about-j-link-gdb-server/
[4] https://www.segger.com/products/debug-probes/j-link/tools/j-link-commander/

Kapitola 8

Závěr

Cílem této práce bylo zjistit, které testy je vhodné vykonávat v zařízeních FPGA a implementovat systém umožňující vzdáleně spouštět tyto testy právě v zařízeních FPGA připojených k centrálnímu serveru. Práce byla zadána firmou Codasip, která implementovaný systém bude využívat.

Software OpenOCD komunikuje přes (externí a integrované) rozhraní JTAG s deskou FPGA a jejím čipem, který se konfiguruje pomocí binárního souboru reprezentujícího procesor. Vhodné testy pro testování procesoru v FPGA jsou takové, které testují způsobem black-box, konkrétně se zde jedná o debugger testy a testy RISC-V debug. Testování na FPGA poskytuje plnohodnotný paralelismus běžících procesů a v některých případech zkrácení doby testů. Systém implementovaný v této práci má architekturu klient-server, která umožňuje vzdáleně operovat s deskami FPGA. Celý navržený systém je implementován v jazyce Python s využitím webového frameworku `flask` a klienta implementovaného pomocí modulu `click`.

Prototyp vyvinutého systému byl aplikován ve firemním testovacím procesu a pomohl zkrátit dobu některých testů až o 22%. Dále umožňuje zaměstnancům firmy vzdálenou obsluhu desek FPGA a zajišťuje k nim rychlý přístup. Tato možnost vzdáleného připojení eliminuje nutnost osobního předávání a půjčování desek. Využitím prototypování pomocí FPGA se firma zbaví části výdajů za licence softwarových simulátorů a uvolní se jí výpočetní zdroje pro jiné testování.

Práce mi přinesla rozšíření znalostí o jazyce Python a některých jeho knihovnách. Dále jsem lépe poznal celý proces vývoje procesoru a testování procesorů ve firmě Codasip. Lépe rozumím deskám FPGA a dokáži s nimi lépe manipulovat a obsluhovat je. Získal jsem celkově lepší přehled o způsobu vývoje digitálních obvodů.

Systém implementovaný v této práci má velký potenciál do budoucna a potřebuje ještě některá rozšíření, například automatické generování souboru bitstream, přidání priority úkolům a možnost zabrat jednu desku více úkoly. Základem dobrého systému jsou testy zajišťující jeho správné fungování při dalším vývoji, tudíž budu pokračovat sepsáním jednotkových a integračních testů. Systém je škálovatelný a v případě potřeby lze některé jeho části nahradit jinými nástroji nebo frameworky tak, aby naplňoval potřeby uživatelů. V budoucnu by se implementovaný systém měl zapojit do právě vznikajícího firemního testovacího systému, což bude vyžadovat udržení vývoje správným směrem.

Literatura

[1] Aldec, I. *HES-DVM Hybrid Verification Platform*. Aldec, Inc. [cit. 2021-04-15]. Dostupné z: https://www.aldec.com/en/products/emulation/hes-dvm--fpga-emulation-platform.

[2] Ashenden, P. J. *The designer's guide to VHDL*. 3. vyd. Morgan Kaufmann, 2010. ISBN 978-0-12-088785-9.

[3] Bakshi, S. a Gajski, D. D. Partitioning and pipelining for performance-constrained hardware/software systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 1999, sv. 7, č. 4, s. 419–432. DOI: 10.1109/92.805749.

[4] Bening, L. a Foster, H. *Principles of verifiable RTL design*. 2. vyd. Springer, 2001. ISBN 978-0-306-47631-0.

[5] Brayton, R. K., Hachtel, G. D. a Sangiovanni-Vincentelli, A. L. Multilevel logic synthesis. *Proceedings of the IEEE*. 1990, sv. 78, č. 2, s. 264–300. DOI: 10.1109/5.52213.

[6] Brown, A. *Nexys A7*. 2021 [cit. 2021-02-20]. Dostupné z: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/start.

[7] Castellano, R. *AMD And Xilinx: The Prize Is Versal ACAP, Not FPGAs (NASDAQ:AMD)*. Oct 2020 [cit. 2020-01-28]. Dostupné z: https://seekingalpha.com/article/4378735-amd-and-xilinx-prize-is-versal-acap-not-fpgas.

[8] Cofer, R. a Harding, B. F. *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. 1. vyd. Elsevier, 2006. ISBN 978-0-7506-7866-7.

[9] Eriksson, M. a Hallberg, V. Comparison between JSON and YAML for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology*. 2011, s. 1–25.

[10] Foster, H. *Part 8: The 2018 Wilson Research Group Functional Verification Study*. January 2019. Dostupné z: https://blogs.sw.siemens.com/verificationhorizons/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/.

[11] Gonzalez, A., Latorre, F. a Magklis, G. *Processor microarchitecture: An implementation perspective*. Morgan & Claypool Publishers, 2010. 1–116 s. ISBN 9781608454532.

[12] Harris, D. *Digital design and computer architecture*. Waltham, MA: Morgan Kaufmann, 2013. ISBN 978-0-12-394424-5.

[13] Hoare, C. A. R. The logic of engineering design. *Microprocessing and Microprogramming*. 1996, sv. 41, 8-9, s. 525–539. DOI: 10.1016/0165-6074(96)00009-9.

[14] Inc., X. *What is an FPGA?* [online]. Xilinx Inc., 2020 [cit. 2020-12-20]. Dostupné z: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[15] International, R.-V. *History - RISC-V International*. Oct 2020 [cit. 2020-12-21]. Dostupné z: https://riscv.org/about/history/.

[16] Kim, D. *FPGA-Accelerated Evaluation and Verification of RTL Designs*. 2019. Disertační práce. UC Berkeley.

[17] Maeda, K. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In: *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*. 2012, s. 177–182. DOI: 10.1109/DICTAP.2012.6215346.

[18] Masařík, K. *What is RISC-V? Why Do We Care and Why You Should Too!* [online]. Codasip s.r.o., Sep 2016 [cit. 2020-12-21]. Dostupné z: https://codasip.com/2016/09/22/what-is-risc-vwhy-do-we-care-and-why-you-should-too/.

[19] Maxfield, C. M. Fundamentals of FPGAs: What Are FPGAs and Why Are They Needed? *Fundamentals of FPGAs*. Digi-Key's North American Editors. Digi-Key's North American Editors. Listopad 2019, [cit. 2020-12-20]. What Are FPGAs and Why Are They Needed? Dostupné z: https://www.digikey.com/en/articles/fundamentals-of-fpgas-what-are-fpgas-and-why-are-they-needed.

[20] Maxfield, C. M. Fundamentals of FPGAs – Part 3: Getting Started with Microchip Technology's FPGAs. *Fundamentals of FPGAs*. Digi-Key's North American Editors. Digi-Key's North American Editors. Leden 2020, [cit. 2020-12-20]. Getting Started with Microchip Technology's FPGAs. Dostupné z: https://www.digikey.com/en/articles/fundamentals-of-fpgas-part-3-getting-started-with-microchip-fpgas.

[21] Mehta, A. B. *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN 3319594176.

[22] Miczo, A. a Lambert, R. D. *Digital logic testing and simulation*. Wiley Online Library, 2003. ISBN 0-471-43995-9.

[23] Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B. et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2016, sv. 35, č. 10, s. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[24] Organization, W. I. P. *Understanding Industrial Property*. Geneva, Switzerland: WIPO, 2016. ISBN 978-92-805-2588-5.

[25] Osborne, A. *Introductions to Microcomputers: Volume On e, Basic Concepts*. McGraw-Hill Osborne Media, 1980. ISBN 9780931988349.

[26] PARTHASARATHI, R. *Computer Architecture*. INFLIBNET Centre, Jul 2018. Dostupné z: http://www.cs.umd.edu/~meesh/411/CA-online/chapter/instruction-set-architecture/index.html.

[27] RASHINKAR, P., PATERSON, P. a SINGH, L. *System-on-a-chip Verification: Methodology and Techniques*. Springer Science & Business Media, 2007. ISBN 0-306-46995-2.

[28] RED HAT, I. *What is a REST API?* Red Hat, Inc. Dostupné z: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[29] REDDY, B. N. K., SURESH, N., RAMESH, J., PAVITHRA, T., BAHULYA, Y. K. et al. An efficient approach for design and testing of FPGA programming using Lab VIEW. In: IEEE. *2015 international conference on advances in computing, communications and informatics (ICACCI)*. 2015, s. 543–548. DOI: 10.1109/ICACCI.2015.7275665.

[30] ROSENFELD, K. a KARRI, R. Attacks and Defenses for JTAG. *IEEE Design Test of Computers*. 2010, sv. 27, č. 1, s. 36–47. DOI: 10.1109/MDT.2010.9.

[31] SAHOO, S. *Do I really need a commercial simulator? - Blog - Company*. Aldec. Dostupné z: https://www.aldec.com/en/company/blog/165--do-i-really-need-a-commercial-simulator.

[32] SEAL, D. *ARM architecture reference manual*. Pearson Education, 2001. ISBN 978-0-201-73719-6.

[33] SELIGMAN, E., SCHUBERT, T. a KUMAR, M. A. K. *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015. ISBN 9780128008157.

[34] VAHID, F. *Digital design with RTL design, VHDL, and Verilog*. 2. vyd. John Wiley & Sons, 2010. ISBN 978-0470531082.

[35] VIJAYARAGHAVAN, S. a RAMANATHAN, M. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2006. ISBN 0-387-26173-7.

[36] WANG, F., WANG, D., YANG, H., XIE, X. a FAN, D. On-Chip Generating FPGA Test Configuration Bitstreams to Reduce Manufacturing Test Time. *Chinese Journal of Electronics*. IET. 2016, sv. 25, č. 1, s. 64–70.

[37] WANG, L.-T., CHANG, Y.-W. a CHENG, K.-T. T. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009. ISBN 9780080922003.

[38] WILSON, R. *Verifying FPGA designs: Simulate, emulate, or hope for the best?* Feb 2009. Dostupné z: https://www.edn.com/verifying-fpga-designs-simulate-emulate-or-hope-for-the-best/.

[39] WOLF, C. Yosys manual. *Retrieved January*. 2021, sv. 16, s. 194, [cit. 2020-12-20].