VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

INTEGRACE FORMÁLNÍCH TECHNIK DO PROCESU VERIFIKACE PROCESORU RISC-V

ENRICHING THE PROCESS OF VERIFICATION OF RISC-V PROCESSOR WITH FORMAL TECHNIQUES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB HORKÝ**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Horký Jakub**
Program: Informační technologie
Název: **Integrace formálních technik do procesu verifikace procesoru RISC-V**
Enriching the Process of Verification of RISC-V Processor with Formal Techniques
Kategorie: Návrh číslicových systémů

Zadání:

1. Nastudujte architekturu RISC-V, problematiku funkční verifikace a metodiky UVM, dostupné formální nástroje a techniky.
2. Analyzujte možnosti užití dostupných nástrojů pro formální verifikaci.
3. Navrhněte tvrzení pro formální verifikaci instrukční sady RISC-V a konzultujte svůj návrh s vedoucím.
4. Implementujte návrh z bodu 3 na příkladu RISC-V procesoru.
5. Analyzujte výsledky a zhodnoťte přínos formálních technik oproti funkční verifikaci.

Literatura:

- Mehta, A. B.: *ASIC/SoC Functional Design Verification*. Springer International Publishing, 2018, ISBN: 9783319866208.
- Mehta, A. B.: *SystemVerilog Assertions and Functional Coverage*. Springer International Publishing, 2018, ISBN: 9783319808338.
- Spear, C. a Tumbush, G.: *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Science & Business Media, 2012, 3rd edition, ISBN: 9781461407157.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Vaňák Tomáš, Ing., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Tato práce krátce rozebírá architekturu RISC-V a návrh procesorů a jak jednoduše může vzniknout chyba při jejich vytváření. Dále popisuji, jakým způsobem se snaží funkční verifikace tyto chyby odhalit a jaké jsou její výhody a nedostatky. Konkrétněji se zaměřím, jak vypadá verifikační prostředí podle UVM. Popisuji, jakým způsobem do funkční verifikace zapadá formální verifikace a jaké jsou dostupné nástroje pro formální verifikaci. Ke konci této práce popisuji konkrétně způsob mého postupu při psaní tvrzení (psaných v SVA jazyce) pro RISC-V procesor za použití nástroje pro formální verifikaci tvrzení. Při využití těchto tvrzení pro ověření procesoru v pozdější fázi vývoje, kdy funkční verifikace již měla možnost většinu chyb odhalit, se mi přesto podařilo několik chyb najít.

Abstract

This thesis provides a brief overview of the RISC-V architecture, design of processors, and how easily a bug can arise during the development. Then this thesis describes the way functional verification tries to discover those bugs and what are its pros and cons. More specifically, the thesis focuses on what the verification environment in UVM look like. Then the thesis describes, how formal verification fits in to the functional verification and shows the tools that are available for formal verification. The final part of this thesis, describes the process of how I wrote the assertions (written in SVA) for a RISC-V processor, using a property checking tool. Using these assertions for verifying a processor in the late stage of development, when functional verification already had the possibility to discover most of the bugs, I still was able to discover few of those bugs.

Klíčová slova

funkční verifikace, formální verifikace, UVM, SVA, tvrzení, RISC-V.

Keywords

functional verification, formal verification, UVM, SVA, assertions, RISC-V

Citace

HORKÝ, Jakub. *Integrace formálních technik do procesu verifikace procesoru RISC-V.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Integrace formálních technik do procesu verifikace procesoru RISC-V

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana profesora Tomáše Hrušky. Další informace mi poskytli Tomáš Vaňák a Jiří Hynek. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Horký
4. června 2020

Poděkování

Chtěl bych poděkovat obvzláště Jiřímu Hynkovi za asistenci při tvorbě práce. Dále bych chtěl poděkovat vedoucímu této práce profesoru Tomáši Hruškovi za rady ohledně prezentování. A Tomáši Vaňákovi za konzultace ohledně praktické části této práce.

Obsah

Kapitola 1

Úvod

Během psaní kódu může velmi jednoduše vzniknout chyba a jelikož komplexita hardwarových designů roste každým dnem, chyb se může objevit mnohem více. Jakým způsobem má hardware fungovat bývá dané specifikací. Jenže specifikace bývají psány normální řečí a tudíž je možné, že programátor, který takovou specifikaci implementuje, špatně pochopí, jak je zamýšlena, což také může způsobit chybu. Historické postupy, kdy bylo možné ručně měnit vstupy a sledovat jaké jsou výstupy a podle toho určit, zda design funguje správně, již nejsou možné. Větší a složitější designy potřebují i sofistikovanější metody ověřování správnosti. Uvedené problémy se snaží řešit verifikace, za pomocí automatizovaných metod.

Mým cílem v této práci je zjistit jakým způsobem funguje funkční verifikace a konkrétně *UVM*. Prozkoumat, jakým způsobem do tohoto procesu zapadá formální verifikace, jaké jsou nástroje a techniky pro práci s formální verifikaci. A nakonec vytvořit tvrzení pro formální verifikaci, které by popisovali procesor s architekturou RISC-V. Ke kontrolování procesoru, který byl použit pro praktickou část této práce, jsem přistoupil až v části, kdy již prošel jinými způsoby funkční verifikace, ale i přesto se mi snad podaří odhalit co nejvíce chyb. Tato práce popisuje informace o postupech používaných k ověřování správnosti hardwaru a může posloužit lidem, kteří se zajímají jakým způsobem funguje technika a baví je hledat chyby v různých postupech a návrzích. Některé způsoby používané pro pro verifikaci hardwaru lze zpravidla použít i pro kontrolu softwaru.

V kapitole 2.2 popisuji, v jakých částech může vzniknout chyba při psaní modelu. Kapitola 3 pojednává o tom, v čem spočívá funkční verifikace. Konkrétněji se zaměřím, jakým způsobem funguje verifikační prostředí podle *UVM* v části 3.2. V kapitole 4 se zabývám, jakým způsobem do tohoto procesu zapadá formální verifikace a dále jaké jsou dostupné nástroje pro formální verifikaci a v čem spočívá práce s nimi. V kapitole 5 se zabývám jakým způsobem psát tvrzení v jazyce *SVA*. V kapitole 6 popisuji můj postup při psaní tvrzení pro procesor založený na architektuře RISC-V. A nakonec v části 6.3 popisuji chyby a nedostatky, které jsem s pomocí těchto tvrzení objevil. Jelikož jsem ale popisoval procesor s architekturou *RISC-V*, tak jsem se nejdříve zaměřil na architekturu *RISC-V* a fungování procesorů v následující kapitole.

Kapitola 2

Procesory

Informace o RISC-V uvedené v této kapitole pochází ze specifikace RISC-V [1, 2]. RISC-V je otevřená ISA (Instruction Set Architecture – instrukční sada) vyvíjená na Californské univerzitě v Berkeley. Uvedená architektura byla původně navržená hlavně pro studijní účely, nicméně postupně se vyvinula v architekturu plnohodnotnou, která lze použít i pro komerční účely.

Jak už vyplývá z názvu, *RISC-V* je založená na principech *RISC* (*Reduced Instruction Set Computer*). Pro architektury založené na principech *RISC*, je charakteristická hlavní myšlenka procesoru, který pracuje s co nejmenším množstvím instrukcí. Tento přístup zjednodušuje velikost a komplexitu procesorů, ale prodlužuje délku programů a přesouvá trochu této komplexity do kompilátorů. Pravým opakem jsou architektury založeny na principech *CISC* (*Complex instruction set computer*). Procesory s architekturou založenou na principech CISC pracují s více instrukcemi, kde některé plní složitější operace, což má za následek kratší programy. CISC kdysi mělo výhodu, že některé komplexní operace mohly být prováděny optimálněji, jenže se ukázalo, že většina programů tyto komplexní instrukce tolik nevyužívala [22] a tudíž spíše zabíraly místo, než aby byly užitečné. Dále existují ještě např. architektury *VLIW* (*Very Long Instruction Word*), kde cílem je, aby jedna instrukce obsahovala více operací, které jsou prováděny paralelně. Tento přístup vyžaduje sofistikovaný překladač, který musí jednotlivé operace správně poskládat do instrukcí během kompilace [3].

RISC-V je navržená pro tvorbu procesorů využitelných v širokém spektru přístrojů. Dají se uzpůsobit k použití v přístrojích s nízkou spotřebou, které nepotřebují velký výkon (RV32EC), nebo třeba i na tvorbu procesorů do stolních počítačů (RV64GC). RISC-V sice u stolních počítačů nemá takovou podporu jako více rozšířené architektury, ale podpora této architektury postupně roste (Linuxové jádro RISC-V podporuje a vznikají i distribuce [10]).

RISC-V je tvořena z několika základních instrukčních sad a rozšířeních, které umožňují procesor přizpůsobit požadavkům na funkčnost. Každý procesor musí implementovat alespoň jednu základní instrukční sadu, protože obsahují základní operace pro práci s celými čísly, s pamětí a pro kontrolu toku programu.

V současné době jsou specifikované čtyři základní instrukční sady: Dvě 32 bitové (RV32E, RV32I), jedna 64 bitová (RV64I) a *ISA* je připravena i pro 128 bitů (RV128I). Všechny instrukce v 32 bitových sadách jsou definovány, aby byly schopny pracovat s jakoukoliv délkou dat. Tudíž instrukční sady s větší délkou dat definují pouze operace pro práci s menšími délkami (pro efektivnější práci s menšími daty – RV64I tedy definuje instrukce pro práci s 32 bitovými daty).

Všechny základní instrukční sady obsahují 32 volně použitelných registrů (kromě *RV32E* se šestnácti registry), kde registr *x0* je speciální registr, který obsahuje vždy hodnotu 0. Tato skutečnost zjednodušuje architekturu v mnoha ohledech (například není potřeba specializovaná instrukce pro přenos dat mezi registry). Tyto registry jsou sice volně využitelné, nicméně ve specifikaci je popsáno, jakým způsobem by měly, být jednotlivé registry použity. Uvedené způsoby není povinné dodržovat, ale jejich správné použití může přinášet nějaké výhody (např. lepší přehlednost kódu, optimalizace některých instrukcí pro práci s určitými registry apod.).

K těmto základním sadám lze přidat různá rozšíření jako třeba násobení, operace pro práci s čísly v plovoucí řádové čárce, instrukce pro zajištění atomičnosti atd. Jednou z výhod RISC-V je, že neomezuje velikost instrukcí. Základní instrukce mají délku 32 bitů, nicméně architektura umožňuje využití rozšíření pro zkrácené instrukce (rozšíření C), které nejčastěji používané instrukce umožňuje zapsat v 16-bitové podobě, a tím zmenšit délku kódu, což zmenší nároky na paměť. Rovněž umožňuje definování vlastních instrukcí, jež mohou být 48-bitové nebo i delší.

RISC-V obsahuje čtyři privilegované módy (Machine, Hypervisor – teprve v návrhu, Supervisor a User). Tyto módy se liší v oprávnění k přístupu ke kontrolním registrům, v možnostech zpracování přerušení a jaké řídící instrukce mohou využívat. Machine mód je povinný pro všechny procesory a má téměř neomezený přístup ke kontrolním registrům. Některé registry omezují přístup všem módům – např. kontrolní registry, které mají povolené pouze čtení. Hodně procesorů implementuje navíc alespoň *user* mód, aby bylo možné zajistit alespoň nějakou ochranu před škodlivým nebo chybným kódem.

K těmto běžně použitelným módům ještě procesor může obsahovat *debug* mód. Procesor se do něj dostane pouze při ladění a má ještě větší pravomoce než *machine* mód (například přístup ke kontrolním registrům *debug* módu). Pro specifikaci způsobu, jakým by měl procesor implementovat *debug* mód, lze využít *RISC-V Debug*, což je specifikace externího ladění určena přímo pro *RISC-V* procesory, ale lze využít i jiné standardní specifikace (např. *Nexus*).

Pro *RISC-V* je výhodné, že jako nově vyvíjená architektura, nemusela být zpětně kompatibilní, a zároveň se může poučit z problémů, které mají některé starší architektury (příprava architektury pro 128 bitů, dostatek prostoru pro nové instrukce apod.). Protože se jedná o otevřenou architekturu, je *RISC-V* zajímavá i pro amatéry (např. lze o značném množství funkcí zjistit, proč jsou specifikované daným způsobem – přímo ve specifikaci bývá uváděno značné množství těchto informací, což způsobuje, že čtení RISC-V specifikace je zajímavější než čtení jiných specifikací).

2.1 Návrh procesoru

RISC-V specifikuje pouze instrukční sadu, což znamená, že popisuje funkce procesoru, ale neurčuje, jakým způsobem mají být prováděny. Během vytváření specifikace byly nicméně respektovány nejběžnější způsoby implementace některých funkcí.

2.1.1 Komponenty v procesoru

Procesor bývá rozdělen do několika logických bloků, kdy každý blok se stará o konkrétní problematiku. Typický procesor může obsahovat například tyto komponenty [21]:

- aritmeticko-logická jednotka (ALU)

- jednotka pro práci s čísly v plovoucí řádové čárce (FPU)
- jednotka pro práci se sběrnicí.
- jednotka načítající instrukce
- mezipaměť
- dekodér
- registrové pole
- řídící jednotka

Nejdůležitější součást každého procesoru představují jednotky, jež se starají o provádění výpočtů. ALU se stará o zpracování aritmetických (např. sčítání) a logických (např. porovnávání) operací nad celými čísly a o výpočty s čísly v plovoucí řádové čárce se stará FPU. Některé procesory ovšem pracují pouze s celými čísly a tedy FPU neobsahují.

Součástí procesorů, také bývá jednotka, která se stará o komunikaci přes sběrnici, ke které je připojeno rozhraní procesoru. Skrz tuto jednotku je prováděna veškerá komunikace s externí pamětí, která bývá také připojena k dané sběrnici. S pamětí jednak potřebují komunikovat některé instrukce procesoru, ale zároveň je potřeba z paměti načítat instrukce, které mají být provedeny. Jelikož komunikace s pamětí je většinou poměrně pomalá, tak procesory často obsahují mezipaměť (*cache*), v které jsou dočasně ukládána dříve načtená/uložená data. Mezipaměť bývá výrazně menší než externí paměť, ale umožňuje dříve použitá data načíst mnohem rychleji. Mezipaměti se používají pro zrychlení komunikace s pamětí, ale pro ukládání mezivýpočtů jsou stále poměrně pomalé a jejich obsah se mění v závislosti na posledních zapisovaných adresách. Tudíž se aktuálně zpracovávaná data ukládají do registrů. Registry jsou dále využívány pro ukládání adres v paměti, informace o současném stavu apod.

Dále procesory obsahují jednotku, která načítá instrukce ještě dříve než je možné tyto instrukce provést. Tato jednotka se snaží předpovídat na jaké adrese se nachází další instrukce. Popřípadě v jednodušších procesorech načte instrukci, která se v paměti nachází po té předchozí. Pokud je adresa následující instrukce předpovězená správně, tak doba mezi prováděním jednotlivých instrukcí je zkrácena. Špatně předpovězená adresa naopak tuto dobu může prodloužit.

Když je instrukce z paměti načtena, musí být dekódována. Dekodér zpracovávané instrukce rozdělí na signály, které jsou použité v dalších jednotkách (typ operace, použité registry apod.).

Jelikož tyto části mohou mít různou rychlost provádění, tak je nutné komunikaci mezi nimi řídit. Kontrolní jednotka se stará o přenos dat mezi jednotlivými komponentami a o jejich synchronizaci.

2.1.2 Optimalizace výkonu

Jelikož tyto komponenty zpracovávají instrukci postupně je možné maximalizovat využití jednotlivých částí za pomocí zřetězeného zpracování (*pipelining*). Při zřetězeném zpracování se provádí více instrukcí zároveň, což umožňuje zvýšit množství provedených instrukcí za jednotku času. Nicméně zřetězené zpracování výměnnou za výkon přináší i řadu problémů. Jeden z problémů představuje čtení po zápisu (jedna instrukce do registru zapisuje a následující z něho čte, tudíž je nutné zajistit, aby druhá instrukce načetla správná data). Dalším

problémem zřetězeného zpracování je, že může zvýšit latenci instrukce (dobu od začátku provádění do konce).

V jednodušších procesorech jsou instrukce prováděny v pořadí, v jakém jsou získávány z paměti, ale často procesory přeskládávají pořadí instrukcí, aby byly ještě lépe využity zdroje. Například, když se čeká na výsledek dělení, což je časově náročná operace, může procesor provádět jiné instrukce, které nevyužívají děličku, nebo nepracují s výsledkem dělení. Nicméně množství instrukcí, které je v procesoru možné přeskládat, bývá poměrně malé, a proto lze lepších výsledků dosáhnout při kompilaci programů [18].

2.1.3 Popis procesoru v programovacím jazyce

Pro psaní programů se využívají softwarové jazyky, které jsou určeny pro psaní sekvenčního kódu, kdy jsou instrukce vykonávány jedna po druhé. Jenže takovéto jazyky se nehodí pro popis hardwaru, kde všechny části probíhají paralelně. proto se pro vytváření hardwaru využívá jazyků na popis hardwaru (např. *Verilog*, *VHDL*). Pomocí těchto jazyků se popisuje model na úrovni registrů (*RTL* – Register Transfer Level). V *RTL* jsou popsány registry a jakým způsobem se mezi nimi přenáší data. Model popsaný v *RTL* je poté během syntézy převeden na jednotlivá logická hradla [23]. Nicméně i tyto jazyky umožňují zápisy kódu, který není možné syntetizovat. Takové části jazyka lze využít například pro vytvoření testovacích prostředí.

2.1.4 Codasip Studio

Jelikož psaní *RTL* pro procesory je náročná práce, tak lze využít programů, které mohou pomoci s jejich vytvářením. Jedním z takovýchto programů je i *Codasip Studio* [5], které lze použít pro vytváření procesorů v jazyce *CodAL* (jazyk vyvíjen firmou *Codasip*). Kromě popisu procesoru ve vyšším programovacím jazyce, z kterého je následně generován *RTL* kód, přináší využití tohoto programu ještě další výhody. *Codasip Studio* navíc totiž umožňuje i využití mnoha dalších nástrojů, které mohou být použity pro jednodušší vývoj procesorů (*Debugger*, kompilátor, simulátor atd.). Kromě toho také generuje základní verifikační prostředí v *UVM*, které lze rozšířit o vlastní verifikaci.

2.2 Vznik chyb

Během vytváření modelu může velmi jednoduše vzniknout chyba. Již jsem se zmínil, že díky rostoucí komplexitě a velikosti designů roste i množství chyb, které se v nich mohou objevit. Je logické, že větší model bude mít i více chyb, ale chyby se objevují i v malých designech.

Nejčastěji jsou chyby způsobeny čistě nepozorností během programování a špatnými praktikami (např. kopírování částí kódu). Velkou část chyb taky představuje špatná komunikace mezi jednotlivými členy týmu, kteří model vytváří. Dalším častým původcem chyb je, když se změní část designu a zapomene se pozměnit jiné závislé části. Dále jsou chyby způsobeny krajními stavy, které při běžném fungování není možné navodit. Chyby se také mohou objevit: v dokumentaci, v návrhu, náhodnou inicializací atd. [20]

Tyto chyby sice přímo nejsou způsobeny tím, že by byl model větší a složitější, ale tato skutečnost usnadňuje jejich vznik. Například větší model tvoří více lidí a tedy roste i pravděpodobnost chyb v komunikaci. Změna v jednom modulu může způsobit chybu v jiném

i čistě z důvodu, že osoba která změnu provádí, neví, že je na něm závislý. Větší model také obsahuje více krajních stavů atd.

2.2.1 Kvalita kódu

Nejčastěji jsou chyby způsobeny špatnými postupy při programování a tudíž nejlepší způsob eliminace chyb představuje psaní kvalitního kódu. Pro zajištění dobré kvality kódu jsou vyvíjeny různé postupy [19] a nástroje, které pomáhají chyby eliminovat.

Názvy proměnných by se měly psát, aby dobře popisovaly jejich využití. Důležité je komentovat složitější části kódu alespoň krátkou informací o tom, k čemu slouží. Stanovení pravidel o psaní kódu (odsazení, délka řádku apod.) zvyšuje přehlednost kódu. Pomocí programu (linter) lze zajistit kontrolu, zda kód splňuje daná pravidla. Linter staticky projde kód a označí všechny jeho části, které nejsou zapsány správným způsobem. Pravidla je možné definovat, nicméně lintery většinou obsahují i vlastní. Lintery ovšem neověří, zda daný kód popisuje zamýšlenou činnost.

Kontrolování kódu

Jedním z nejlepších způsobů eliminace chyb je, aby každou část kódu, kterou někdo napíše, prošel i někdo další. Tato další osoba si udělá vlastní průzkum o dané problematice a kód si otestuje. Uvedený způsob bývá doveden do extrému při párovém programování, kdy jeden člověk programuje, druhý sleduje a kontroluje (využívá se např. v extrémním programování).

Kapitola 3

Funkční verifikace

Informace o funkční verifikaci uvedené v této kapitole pochází z této knihy [4] a z kurzů na stránce VerificationAcademy [14] (kromě kurzů zde lze nalézt fóra a články související s verifikací).

Funkční verifikace se snaží zajistit, aby hardware fungoval stejným způsobem, jaký je uvedený ve specifikaci. Verifikační proces je ovšem velmi dlouhý (dokonce představuje část delší, než samotná tvorba designu) a zároveň bývá náročný na zdroje.

3.1 Testování

Nejčastější způsob ověřování správnosti představuje testování. Kód procesoru je spuštěn (nejčastěji v *simulaci*, ale je možné například využít *FPGA*) a na vstup jsou posílána data, která mají ověřit správnou funkčnost. Testování probíhá pokud možno od prvních částí vývoje, aby případná chyba byla odhalena co nejdříve. Model musí být znovu otestován s každou provedenou změnou a to i na vstupy, které již dříve testované byly a zdánlivě s touto změnou nesouvisejí (objevení chyby brzo po provedených změnách většinou značí, že buď tyto změny obsahují chybu, nebo odhalují chybu v jiné části, což může zjednodušit její lokalizaci). Všechny testy by měly mít předem definovaný očekávaný výstup, aby bylo možné pouze kontrolovat, zda test prošel bez komplikací.

3.1.1 Přímé testy

Základní testování je prováděno za pomocí přímých testů, které jsou vytvořeny se záměrem otestovat konkrétní činnost nebo sekvenci. Bývají psány designéry, kteří implementovali komponentu a potřebují otestovat její základní funkčnost nebo situaci, jež by podle nich mohla být problematická. Zároveň je důležité, aby vždy správnou funkcionalitu testoval i jiný člověk, než pouze její autor, protože se může stát, že si autor špatně vyloží specifikaci, nebo na něco zapomene.

3.1.2 Náhodné testy

Jelikož není možné napsat přímé testy takovým způsobem, aby pokrývaly všechny možnosti, jaké mohou na vstupu nastat, tak se využívá náhodně generovaných vstupů. V případě procesorů dochází ke generování náhodných programů a kontroluje se, zda program proběhne v pořádku. Generování se omezuje na vstupy, které mají jasný a konečný cíl, aby bylo zajištěno, že program proběhne do konce. Například při testování paměti je vhodné generovat

náhodná data a adresy s následnou kontrolou, zda paměť obsahuje správná data na daných adresách. Některé jazyky (např. SystemVerilog) usnadňují generování náhodných dat prostřednictvím definování proměnných jako náhodných a zároveň omezením, jakých hodnot mohou nabývat.

3.1.3 Referenční model

Implementace referenčního modelu, představuje jeden ze způsobů kontroly správnosti výsledků. Nemusí řešit optimalizace, tedy i problémy, které přináší (stačí, aby došel ke správnému výsledku). Může být naimplementován také jako hardwarový design, nebo třeba jako program v nějakém vyšším programovacím jazyce (pro RISC-V existují open source simulátory např. SPIKE).

3.1.4 Pokrytí

Informace o pokrytí (*Coverage*) se využívají, aby bylo možné zjistit, jaké možnosti nastaly při spouštění testů. Pokrytí kódu (*Code coverage*) představuje základ, který pouze kontroluje, zda byly využity všechny části kódu. Jaké konkrétní pokrytí kódu je v simulátorech podporováno, bývá uvedeno v dokumentaci. Ale celkově existuje poměrně hodně typů [15]:

- Pokrytí změn kontroluje, zda všechny bity registrů nebo signálu byly změněny z 0 na 1 a naopak.
- Pokrytí řádků kontroluje, zda byly pokryty všechny řádky v kódu.
- Pokrytí příkazů kontroluje, zda byly pokryty všechny příkazy v kódu. Tato metrika bývá užitečnější než pokrytí řádků, protože příkaz může být přes několik řádků, nebo naopak řádek může obsahovat více příkazů.
- Pokrytí bloků kontroluje, zda všechny bloky v kódu byly použity. Blok je definován jako několik příkazů za sebou, kde když se provádění dostane na začátek bloku, tak se provedou všechny příkazy v něm.
- Pokrytí větví kontroluje, zda u podmíněných výrazů byly použity všechny větve.
- Pokrytí výrazů kontroluje, zda všechny Booleovské výrazy byly někdy pravdivé a někdy nepravdivé. Pokud je výraz složený z více podvýrazů oddělených logickým operátorem, tak jsou kontrolovány všechny tyto podvýrazy
- Zaměřené pokrytí výrazů (*FEC*) představuje silnější kontrolu pokrytí výrazů v podobě kontroly, že každá část podmínky, měla přímý vliv na výsledek rozhodnutí a že výsledek rozhodnutí se dostal na všechny možnosti.
- Pokrytí stavových automatů zkoumá, zda se stavové automaty dostaly do všech stavů a zda byly využity všechny přechody, mezi jednotlivými stavy. Popřípadě umožňuje i pokrytí cest ve stavovém automatu.

Mnoho z těchto typů měří téměř stejnou funkcionalitu popřípadě rozšiřuje některé jiné. Protože pokrytí může značně prodloužit dobu simulace, tak se nepoužívají všechny tyto typy zároveň (například pokrytí bloků může obsáhnout pokrytí větví, příkazů i řádků).

Pokud nějaká část v kódu není pokrytá, je potřeba zjistit z jakého důvodu. Část může být nepokrytá, protože okolnosti, za jakých by se design dostal do daného stavu není dostatečně lehké navodit, aby je mohly odhalit náhodné testy. V tomto případě vyvstává nutnost

zajistit její otestování přímým testem. Část kódu může být nepokrytá, protože představuje chybu v designu a neexistuje způsob, jakým by se dalo k dané části dojít. Poté musí být zjištěno, zda je kód zbytečný, tedy by měl být odstraněn, nebo chyba vznikla v nějaké jiné části (např. špatně definovaná podmínka, která přináší vždy stejný výsledek). Při hledání uvedeného problému může pomoci, že mnoho chyb může být odhaleno pomocí vícero druhů pokrytí (např. když není pokryta určitá větev a zároveň se řídící signál nikdy nedostal do hodnoty, která by to umožnila.).

Nicméně pokrytí kódu má značnou nevýhodu, a to že kontroluje pouze implementovanou funkcionalitu, a tudíž je spíše užitečné ze začátku tvorby hardwaru, protože může značně urychlit hledání chyb. Ale aby byla zajištěna kontrola všech možností a testování kompletní funkcionality, musí být pokrytí doplněno o pokrytí vlastní. Vlastní pokrytí může například obsahovat informace o tom, zda se design dostal do nějakého krajního stavu nebo s jakými hodnotami byl testován apod. Pokud má testovaný design více funkcí je užitečné zjišťovat, zda byly otestovány všechny jeho funkce (třeba použití všech instrukcí). Uvedené pokrytí také samozřejmě mohou být různě kombinovány (třeba všechny instrukce byly použity s krajními hodnotami). Toto pokrytí by šlo ještě rozšířit, že byly použity všechny kombinace po sobě jdoucích instrukcí (nastala situace, kdy byla instrukce A před B a naopak). Podobným způsobů pokrytí by bylo možné si vymyslet neomezené množství, ale jak jsem již zmínil, tak pokrytí může poměrně zpomalovat simulaci, takže záleží do jaké míry chceme kontrolovat správnou činnost.

3.1.5 Tvrzení

Důležitou část sebe-kontrolujícího testování představuje tvrzení o modelu. Pokud o nějakém signálu můžeme říct nějaké tvrzení je možné je i zapsat (způsoby psaní tvrzení v jazyce *SVA* popisuju v části 5). Tvrzení lze například vhodně využít pokud nesmí nastat situace, kdy se dva signály na výstupu budou rovnat. Dále je vhodné použít tvrzení, že na vstupu modulu se neobjeví kombinace signálů, která není podporovaná (používá se i v softwarovém programování, například že argumenty funkce obsahují správnou hodnotu)

Použití tvrzení v simulaci s sebou přináší několik výhod. Může usnadnit lokalizaci chyb, když nastane chyba na výstupu a zároveň selže nějaké tvrzení. Tvrzení samy mohou odhalit chybu, když selže tvrzení, které se neprojeví na výstupu. Další výhodou může být, že samotné psaní tvrzení může pomoci odhalit nějaké chyby, pokud tvrzení píše programátor, který vytvořil danou funkcionalitu (může si uvědomit chybu ve svém návrhu).

3.2 UVM

Použité informace o *UVM* pocházejí převážně z kurzů a článků na VerificationAcademy [14], ale využíval jsem ještě stránky Duolos [12] a ChipVerify [13]. Pro jednodušší pochopení jsem také využil *UVM beginners guide* [16].

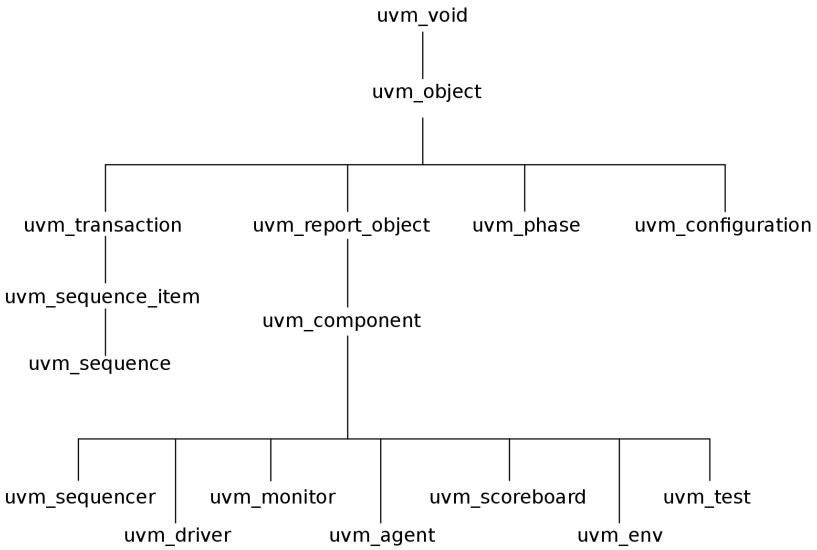
Pro rychlou a efektivní kontrolu hardwaru jsou vyvíjeny všemožné postupy, které mají zvýšit kvalitu a znovupoužitelnost kódu, což umožňuje snížení doby a nákladů potřebných pro tvorbu verifikačních prostředí. *UVM* (*Universal Verification Methodology* – univerzální verifikační metodika) představuje jeden z takových postupů. První verze *UVM* byla vydána v roce 2011 jako standard organizací *Accelera*, která se zaměřuje na automatizaci tvorby elektronických designů. *UVM* vychází převážně z *OVM* (Open Verification Methodology – Otevřená verifikační metodika), na jejímž vývoji se podílelo několik firem, které se zabývají verifikací hardwaru. *UVM* je implementované jako knihovna pro *SystemVerilog* a podporují

ho mnohé firmy, které vyvíjí simulátory pro jazyky na popis hardwaru. *UVM* je dodáváno společně se simulátory, případně je možné si jej stáhnout ze stránek organice *Accelera*.

UVM představuje objektově orientovanou metodiku, jejímž hlavním cílem je znovupoužitelnost verifikačního prostředí, proto obsahuje mnoho způsobů jednoduché obměny použitých částí. Značné změny funkcionality testovacího prostředí lze dosáhnout pouhou změnou několika řádků kódu, nebo jen parametrem při spouštění testu.

3.3 Komponenty

UVM implementuje několik tříd, z kterých jednotlivé implementace dědí. Tyto třídy lze do verifikačního prostředí v jazyce *SystemVerilog* naimportovat jako balíček pomocí *import uvm_pkg::**. Stromová struktura nejdůležitějších tříd je na obrázku 3.1.



Obrázek 3.1: Stromová struktura nejdůležitějších tříd v *UVM*. [1]

Sekvence

Pro každý test jsou potřebná testovací data, o což se starají sekvence. Sekvence vytvářejí data, která mají otestovat jednu základní činnost testovaného zařízení (při testování *ALU* jedna sekvence řeší sčítání, druhá odčítání apod.). Sekvence dědí z komponenty *uvm_sequence*.

Řadič

Řadič (sequencer) se stará o získávání dat ze sekvencí a posílá je dál.

Driver

Většinou se při vytváření dat nechceme zabývat způsobem jejich přenosu do testovaného modelu (sériově/paralelně, jaký protokol bude použit apod.). Ke komunikaci se využívá

[1] Obrázek převzat z https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/

komponenty *driver*. *Driver* představuje komponentu dědící z *uvm_driver*, implementuje funkce k získání dat z řadiče a stará se o převod těchto dat na formát, který bude moci přijmout testované zařízení.

Transakce

Jelikož se mimo komponenty *driver* nechceme zabývat, jak data vypadají při komunikaci s testovaným zařízením, proto se pro komunikaci mezi jednotlivými komponentami využívají transakce. Transakce většinou dědí z *uvm_transaction* nebo *uvm_sequence_item* a je použita jako základní nosič dat v *UVM*. Zajišťuje jistou úroveň abstrakce. Transakce obsahuje proměnné s jednotlivými daty a případně pomocné funkce pro jejich interpretaci (převod na text apod.).

Monitor

Monitor funguje přesně opačným způsobem než *driver*. Analyzuje signály, které jsou na portech testovaného zařízení, a převádí je na transakce. *UVM* neomezuje množství monitorů obsažených ve verifikačním prostředí, lze například použít jeden monitor pro získání dat ze vstupu a druhý pro data z výstupu. Data ze vstupu lze použít například k zjištění, jaká data byla otestována. Data z výstupu může umožnit analýzu správné reakce designu.

Agent

Agent představuje komponentu, která za běhu simulace neprovádí žádnou činnost, ale používá se k inicializaci jednotlivých komponent. Agent nejdříve vytvoří jednotlivé komponenty a poté je propojí. Připojí k sobě *řadič* a *driver*, výstup *driveru* ke svému výstupnímu rozhraní, vstup monitoru ke svému vstupnímu rozhraní a výstup monitoru na svůj analyzační výstup.

Scoreboard

Scoreboard je velmi důležitá část, každého automatického sebekontrolujícího testování, protože analyzuje data získaná z monitorů. Analyzuje informace získané z monitorů a kontroluje správnost výstupů. Po doběhnutí simulace můžeme díky datům získaných ze scoreboardu zjistit míru funkčnosti designu. Podobně, jako může být více monitorů, může být i více scoreboardů, každý kontrolující jinou činnost.

Prostředí

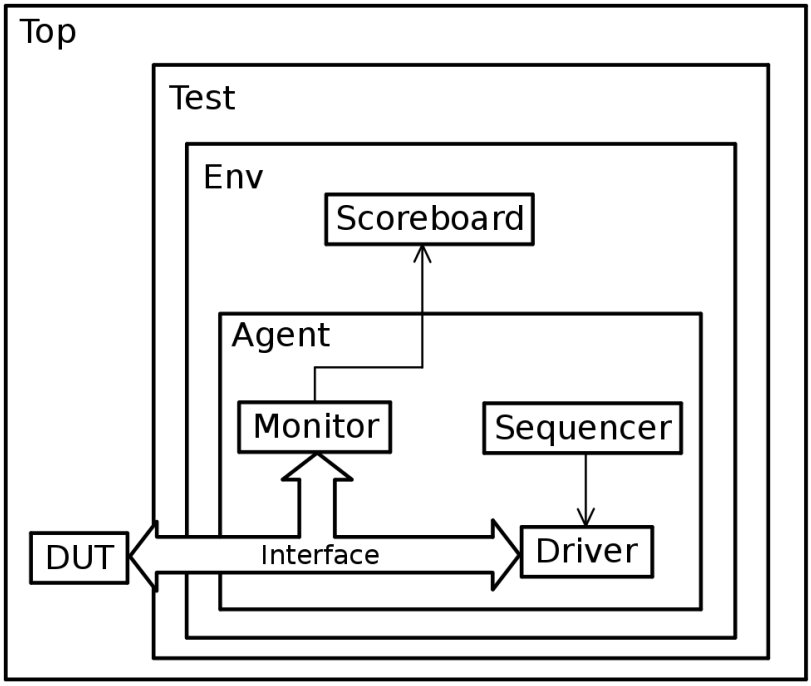
Prostředí (environment) plní obdobnou roli jako *agent*. Vytváří *agenta* a scoreboard. Propojuje je a připojuje vstup/výstup agenta na svoje vstupy/výstupy.

Testbench

Všechny uvedené komponenty jsou obsaženy v komponentě *testbench*. *Testbench* je místo, v kterém se určí, jaké druhy komponent budou použity, pokud verifikační prostředí obsahuje možnosti testování designů s různými vstupní rozhraními, funkcemi apod. Zároveň opět zajistí propojení svých vstupů/výstupů na vstupy/výstupy prostředí.

Top modul

Top modul již kromě komponenty *testbench* obsahuje i zařízení, které má testovat. Připojí výstup z této komponenty na vstup zařízení a naopak. Výsledné prostředí může vypadat například jako na obrázku 3.2.



Obrázek 3.2: Verifikační prostředí podle UVM se skládá z několika komponent, které zajišťují každá jinou samostatnou činnost. Komponenty nižší úrovně se starají o zpracovávání dat, zatímco komponenty na vyšší úrovni zařizují jejich vytváření a propojení.[3]

3.3.1 Porty

Komponenty v *UVM* pro komunikaci mezi sebou využívají porty. Komponenty neví o tom, kolik dalších je připojeno na jejich výstupní port, a vědět to nepotřebují, protože *UVM* se o komunikaci postará za ně. Tím se usnadňuje komunikace, protože komponenty pošlou data na výstupní port a už se nestarají o způsob zpracování nebo zda vůbec došlo ke zpracování (zajišťuje nadřazená komponenta). Komponenty posílají data na svůj výstupní port pomocí operace *write* a všechny připojené komponenty musí tuto operaci implementovat. Při vložení dat na výstupní port je zavolána operace *write* u každé z připojených komponent.

3.3.2 Továrna

Továrna (*factory*) představuje poměrně známý koncept používaný v objektivně orientovaném programování a je i důležitou součástí *UVM*. Jednotlivé komponenty se při své definici zaregistrují do továrny. Pokud je potom v nějaké jiné části tato komponenta vyžadována, zadá se továrně, aby ji vytvořila. Výhodu uvedeného postupu představuje možnost, aby

[3]Obrázek převzat z https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/

továrna vytvořila komponentu jinou, což umožňuje jednoduše měnit činnost verifikačního prostředí.

3.3.3 Fáze

Život jednotlivých komponent v *UVM* je rozdělen do několika částí, které se ještě dělí do jednotlivých fází. Každá komponenta implementuje tyto fáze ve formě metod. Ty jsou postupně volány v jednotlivých částech simulace, tím jsou komponenty synchronizovány. Implementování všech fází není povinné, a proto většina komponent implementuje jen několik fází.

Inicializační část

Před začátkem simulace proběhne stavební fáze, během které jsou jednotlivé komponenty vytvořeny. Poté následuje propojovací fáze, v jejímž průběhu jsou jednotlivé komponenty propojeny. Při vytváření komponent se postupuje od nadřazených komponent k nižším. Při připojování komponent se postupuje opačným způsobem. Následně ještě proběhne konečná fáze, kterou lze využít pro případné drobné úpravy.

Běhová část

Před začátkem simulace se spouští předsimulační fáze, která probíhá z dola-nahoru. Čehož lze využít například k vytisknutí topologie, nebo různých informací o nastavení verifikačního prostředí. Dále následuje běhová fáze, která je nejdelší ze všech fází, protože je aktivní od začátku simulace do konce. V této fázi všechny komponenty běží paralelně a probíhá v ní testování designu a sbírání dat o modelu.

Konečná část

Po skončení simulace proběhnou všechny fáze, které analyzují získaná data. V první fázi jsou extrahována získaná data ze *scoreboardu*. V další fázi jsou získaná data zpracována a zjišťuje se míra funkčnosti designu. Případně se identifikuje v jakých částech došlo k chybě. Poté následuje report fáze, která se používá k vypsání výsledků nebo uložení do souboru. Finální fáze obsahuje vše zbývající, co chceme před koncem simulace provést.

3.3.4 UVM makra

UVM využívá továrnu, proto musí být definovaný mechanismus, prostřednictvím kterého se v ní jednotlivé komponenty mohou registrovat. Pro tento účel *UVM* implementuje makra. Tyto makra lze do verifikačního prostředí v jazyce *SystemVerilog* naimportovoat pomocí ‘`include "uvm_macros.svh"`. Třídy derivované z `uvm_object` nebo `uvm_transaction` využívají pro registraci ‘`uvm_object_utils`. Komponenty dědící z `uvm_component` se registrují pomocí ‘`uvm_component_utils`. Obě tyto makra se používají s jedním argumentem, třídou, v které jsou použity. Pokud jsou třídy parametrizovány, měly by využít např. ‘`uvm_object_param_utils`. Při vytváření těchto tříd je potřeba místo volání metody `new` použít `type_id::create`, která používá továrnu k vytvoření objektu. Továrna zavolá metodu `new` u objektu, jenž má definovaný místo požadovaného objektu.

UVM dále implementuje automatické funkce pro tisk, kopírování, porovnávání. Těmto funkcím lze dodat implementaci v podobě funkcí `do_print`, `do_copy`, `do_compare`. Pokud se během definice proměnných využije makra ‘`uvm_field_*` (* se nahradí požadovaným typem

– třeba int), potom není nutné tyto funkce definovat. Lze využít obě možnosti současně, některé proměnné nechat tisknout automaticky a k některým dodat implementaci (vhodné u tříd, které chceme tisknout specifickým způsobem). ‘`uvm_field_*` makra přijímají jako argument název proměnné a příznaky (*flag*), prostřednictvím kterých lze tyto proměnné vyjmout z některých funkcí.

UVM ještě implementuje několik maker pro tisk informací. Hlavní výhoda těchto maker oproti použití metody ‘`$display` apod. spočívá ve skutečnosti, že tisknou dodatečné informace, které mohou pomoci s laděním (informace o simulačním čase a řádku a souboru z něhož jsou volány). Každé z těchto maker má definovaný význam a výchozí chování, které lze přepsat. Těmto makrům kromě zprávy, jakou mají vytisknout, je nutné specifikovat ID (nemusí být jedinečné, lze je tedy shlukovat k sobě např. v rámci jednoho testu). Na konci simulace se tiskne, v jakém množství se všechna tato makra a jednotlivé ID objevily.

Makro ‘`uvm_info` se používá k tisku informací (hlavně ladící informace). Při definici tohoto makra je třeba uvést úroveň verbosity. Pro definování verbosity lze využít předem definovaných hodnot (`UVM_NONE`, `UVM_LOW`, `UVM_MEDIUM` – výchozí, `UVM_HIGH`, `UVM_FULL`, `UVM_DEBUG`), kde nejnižší se vytiskne vždy a nejvyšší téměř nikdy. Pro určení tisknutelných úrovní lze využít např. funkci `set_report_verbosity_level(UVM_LOW)`, která nastaví nejvyšší tisknutelnou hodnotu verbosity dané třídy.

‘`uvm_warning`, ‘`uvm_error`, ‘`uvm_fatal` jsou makra, která jsou použita k vytisknutí problémů s různou vážností. ‘`uvm_fatal` kromě vytisknutí informací ještě ukončí simulaci. Tisknutí informací z těchto maker nelze upravit pomocí změny limitu verbosity ale lze upravit akci, která nastane při použití daného makra. Úpravu akce pro danou vážnost je možné provést pomocí `set_report_severity_action`.

Kapitola 4

Formální verifikace

Kurzy dostupné na VerificationAcademy [14] jsem opět využil pro získání základních informací o formální verifikaci.

UVM a další metody funkční verifikace, které využívají simulaci, mají jednu značnou nevýhodu. Za použití kombinace náhodných a přímých testů není možné pokrýt všechny možnosti, které mohou na vstupu nastat. A tudíž se využívá dalších metod než simulace k ověření správnosti.

Formální verifikace je proces, během kterého se za využití matematických důkazů zjišťuje, zda daný systém splňuje požadavky, které jsou na něj kladeny specifikací. Všechny formální nástroje analyzují model (rozdělují jednotlivé registry na stavový automat) a zjišťují, zda splňuje všechny požadavky. Nicméně většina formálních nástrojů nepracuje s hodnotou 'X' (nedefinovaná hodnota). Signál, který by v simulaci bylo možné nastavit na hodnotu 'X' je použitý jako kontrolní bod, což může způsobit odlišnost formálních/simulačních výsledků.

4.1 Techniky

Nevýhodu formálních nástrojů představuje problém se systémy, které obsahují velké množství stavů, protože komplexita vytvoření důkazu razantně roste s množstvím stavů, jež mohou nastat. Při definici tvrzení, které kontrolují velké množství stavů, roste pravděpodobnost, že čas potřebný k jeho potvrzení bude výrazně delší, než je žádoucí (a potencionálně nekonečný), i když bude tvrzení platné. Tudíž vzniká potřeba definování tvrzení o modelu takovým způsobem, aby kontrolovala co nejméně stavů. Nicméně množství stavů lze omezit i dalšími způsoby, než zjednodušováním tvrzení.

4.1.1 Omezení stavů

Pro zjednodušení komplexity pro formální nástroje lze specifikovat, aby nevyužíval některé hodnoty na vstupech, nebo aby nekontroloval některé stavy. Tento přístup může přijít k užitku, pokud chceme kontrolovat specifickou činnost/funkci. Kontrolu specifické činnosti chceme většinou použít v případě, pokud byla naimplementována nová funkčnost a my ji chceme ověřit co nejrychleji, nebo hledáme chybu a víme, co zhruba chybu způsobuje. Danou funkčnost by bylo samozřejmě možné ověřit i bez omezení vstupu, ale jakýkoliv způsob, který nám pomůže chyby odhalit rychleji je užitečný.

Pokud program objeví chybu při omezení vstupů, znamená to, že chybu nalezl správně. Nicméně v případě omezeného vstupu, nemusí být nalezené důkazy pravdivé – jejich pravdivost platí pouze v případech, na které je vstup omezen.

Pokud nehledáme konkrétní chybu a chceme kontrolovat správnou činnost celku, jeví se lepším postupem vstup moc neomezovat. Může nastat situace, že funkce, kterou považujeme za neovlivněnou omezením vstupů, tudíž na ni nahlížíme jako na bezchybnou, ve skutečnosti obsahuje chybu, která je těmito předpoklady maskována.

4.1.2 Černá skříňka

Při použití formálních nástrojů může vzniknout potřeba omezení velikosti stavového prostoru, tudíž se jeví užitečným některé komponenty z kontroly vyjmout. Nejvhodnější je vyjmutí komponent, které přímo nesouvisejí s kontrolovanou záležitostí. Pokud kontrolujeme způsob zpracovávání dat získaných z paměti modelem, není potřeba ověřovat správné fungování k tomu připojené paměti. Pokud paměť zapouzdříme do černé skříňky, výstup paměti se stane řídícím bodem pro formální nástroj. Nicméně komponenty zapouzdřené do černé skříňky musí být verifikovány zvlášť.

Za předpokladu, že komponenta zapouzdřená v černé skříňce neobsahuje chybu, jsou získané důkazy i protipříklady rovněž správné. Pokud komponenta v černé skříňce chybu obsahuje, potom získané důkazy/protipříklady nemusí být správné.

4.2 Nástroje

Informace o nástrojích jsem získal ze stránek výrobců: *Cadence* [6], *Mentor* [8], *Synopsys* [7].

Formální nástroje se dají rozdělit na tři hlavní typy přičemž většina programů implementuje více než jeden typ.

4.2.1 Kontrola ekvivalence

Nástroje pro kontrolu ekvivalence analyzují dva modely a zjišťují zda jsou ekvivalentní. Existují dva druhy nástrojů pro kontrolu ekvivalence. Nástroj kontrolující, zda:

- oba modely obsahují stejné stavy,
- vstup vyvolá na obou modelech stejný výstup.

Aby bylo možné ověřit větší designy, které obsahují velké množství stavů (jako procesory), vzniká potřeba analýzy na nižších úrovních. Nejprve se zkontrolují jednotlivé moduly, zda vykonávají stejnou funkci. Pokud jsou totožné, označí se jako stejné. Poté již není nutné, aby jejich stavový prostor byl součástí celkového stavového prostoru.

Model popsaný na úrovni registrů se během syntézy převede na model na úrovni jednotlivých logických hradel. Nástroje, používané na syntézu sice funkčnost modelu nemění, ale bohužel v podstatě každý program obsahuje nějakou chybu, mohlo by se tedy stát, že během syntézy a následných optimalizací se výsledná logika trochu změní. Tudíž se kontrola ekvivalence nejčastěji používá na zjištění, zda syntéza *RTL* proběhla v pořádku, kdy se porovnává původní *RTL* model s následným syntetyzovaným modelem.

Když se blíží konec vývoje modelu a správná funkčnost je v podstatě ověřena, začínají se provádět různé optimalizační úpravy. Kromě optimalizace rychlosti a plochy ještě často probíhají optimalizace spotřeby. Některé optimalizace vypínají části modelu, které nejsou pro současnou činnost potřebné, popřípadě upravují způsob nastavování signálů v případech, kdy jejich výsledek není potřeba. Při provádění změn, které neupravují funkci designu, lze kontrolu ekvivalence využít pro ověření, zda změna provedená v designu neměla vliv na funkčnost.

Dále lze využít kontrolu ekvivalence pro modely, které jsou navrženy ve více *HDL* (Hardware Description Language – jazyk na popis hardwaru). Hlavní přínos ověření ekvivalence v tomto případě spočívá ve skutečnosti, že stačí verifikovat komponentu pouze v jednom jazyce. V druhém stačí pouze zajistit její totožnost, což je méně náročné než verifikovat oba jazyky zvlášť.

4.2.2 Kontrola tvrzení

Vstup těchto nástrojů představuje model a tvrzení, které jej popisují. Program poté každé tvrzení o modelu prověří, buď dokáže jeho pravdivost, nebo vytvoří protipříklad, který dané tvrzení vyvrací. Využití tohoto nástroje vyžaduje poměrně dost času a vědomosti o psaní formálních tvrzení, protože program potřebuje jako vstup model a tvrzení o něm. Tyto nástroje jsou vhodné pro použití ke kontrole zamýšlené funkčnosti (správnosti výsledků), i na kontrolu časových operací (např. že komponenta vrátí výsledek do 4 hodinových taktů).

4.2.3 Automatická analýza modelu

Použití nástroje pro kontrolu tvrzení ovšem bývá využito i jinými programy, které automaticky analyzují model a snaží se najít různé problémy, jež mohou v designu nastat. Těmto nástrojům většinou stačí model a informace o kontrolované problematice. Z nich se automaticky vygenerují formální tvrzení. Popsané nástroje jsou velmi užitečné obzvlášť v počátku vytváření hardwaru, protože mohou odhalit mnoho chyb dříve, než je dostupné verifikační prostředí pro funkční verifikaci, nebo popis vlastností pro formální analýzu. Většina programů obsahuje více než jeden způsob automatické analýzy.

Kontrola zablokování/zacyklení

Program zanalyzuje model a snaží se najít způsob, díky němuž se model ocitne ve stavu, z kterého by nebylo možné se dostat do stavu konečného. Uvedená možnost může nastat, pokud se model zablokuje ve stavu, z něhož by se mohl dostat pouze v případě, že by nastala možnost, která ale nastat nemůže (*deadlock*). Zacyklení představuje druhý způsob, jak tato situace může nastat. Tehdy model prochází stejnými stavy v cyklu, ze kterého se nelze dostat (*livelock*).

Kontrola pokrytí

Pro pokrytí kódu za použití simulace představuje problém dosáhnout 100% pokrytí, a proto existují formální nástroje, které tento problém řeší. Programu se předají data o pokrytí získané ze simulace a zanalyzuje model, aby našel způsob pokrytí nepokrytých částí. Výstupem uvedeného nástroje může být ukázka způsobu pokrytí určité části, což lze využít na aktualizaci funkční verifikace. Dále může přinášet informace, že pokrytí nelze dosáhnout ani pomocí formálního nástroje, což znamená, že by bylo lepší danou část zkontrolovat. Popřípadě výstupem může být informace, že určitou část kódu pokrýt nelze, jedná se tedy o mrtvý kód.

Kontrola registrů

Součástí procesorů jsou kontrolní registry, které mívají různá omezení (některé mají zapisovatelné pouze některé bity, jiné jsou pouze pro čtení atd.). Pokrytí všech možností pomocí

simulace bývá složité a proto lze využít formální nástroj pro kontrolu registrů. Pro použití takového nástroje samozřejmě musí být definované informací o jednotlivých registrech.

Kontrola 'X' hodnot

Hodnota 'X' znamená, že není známá (např. pokud se čte z neinicializované paměti). Problém nastává v interpretaci 'X' hodnot: V reálném systému signál sice obsahuje neznámou hodnotu, ale stále nějakou nese, takže výsledek představuje sice neznámá reakce, ale uvedené by stejně nemělo způsobit chybu v designu. K problémům dochází během simulace. Když se v nějaké části designu objeví 'X', může se stát, že se dostane do kritické části, kde způsobí chybu. Jelikož 'X' hodnota není známá, je výsledek porovnání vždy nepravdivý (v jazyce *Verilog*). Když se hodnota 'X' dostane do kontrolního signálu, který se používá na podmíněné přiřazení více signálů, může se stát, že se nastaví signály, jež by společně za jiných okolností nikdy nemohly nastat (např. kdyby se dostala hodnota 'X' do signálu značící chybu). Uvedený problém může v simulaci způsobit chybně negativní výsledky. Pro řešení zmíněné problematiky lze využít plně automatického formálního nástroje.

Kontrola bezpečnosti

Pokud design obsahuje zabezpečená data, je vhodné zjistit, zda nemůže nastat situace, kdy by se data dostala do nezabezpečené části (např. hodnota kontrolních registrů se nedostane na výstup, když nemá). Zároveň se nesmí stát, aby data byla upravena, pokud změna nemá nastat. Chybná změna může mít původ v použití designu jiným způsobem, než bylo zamýšleno, nebo z důvodu náhodného přepsání kvůli chybě v designu. Formální nástroje kontrolující bezpečnost potřebují model a definici zabezpečených částí a způsobů, prostřednictvím nichž lze k těmto částem přistupovat.

Kontrola propojení

Ruční kontrolování správnosti propojení všech modulů představuje náročnou a hlavně zdlouhavou práci. Vytvoření testovacího prostředí také klade velké nároky na čas. A proto existují formální nástroje řešící tento problém. Takovému programu stačí předat *RTL* a specifikaci propojení signálu v podobě tabulky (XML). Ve zmíněném případě dochází k ignorování všech modulů a formální analýza se soustředí pouze na jejich propojení.

Kontrola spotřeby

Spotřeba představuje důležitou problematiku každého hardwaru, kterou nelze opomenout. V mnoha částech designu lze udržet stejný výkon s nižší spotřebou, proto je žádoucí spotřebu redukovat. Formální nástroj kontrolující spotřebu potřebuje *RTL* a specifikaci o spotřebě.

Kontrola hodinových domén

Některé designy mohou obsahovat více částí, z nichž každá může být závislá na jiném hodinovém signálu, což může způsobovat problémy při společné komunikaci jednotlivých části. Ověřování správné činnosti pomocí simulace může být poměrně náročné, ale naštěstí opět existuje automatický formální nástroj řešící zmíněný problém, a to pouze s *RTL* jako vstupem.

4.3 Použitý program

Rozhodl jsem se pro využití nástroje pro formální kontrolu tvrzení. Hlavním důvodem bylo, že zadání práce to víceméně určuje, a také některé další nástroje již byly použity. Formální nástroje na kontrolu hardwaru jsou vyvíjeny firmami, které se zabývají verifikací (*Cadence*, *Mentor*, *Synopsis*). Nicméně existuje i *open source* program (*yosys*), ale ten nenabízí dostatek funkcí, aby splňoval hledisko použitelnosti pro komerční účely.

Konkrétně jsem se rozhodl použít program *Questa Formal* od společnosti *Mentor*, což je společnost nabízející velkou řadu nástrojů používaných k verifikaci. Kromě formálních nástrojů vyvíjí třeba simulátor a verifikační IP pro některé často používané protokoly. Výhodu použití *Questa Formal* představuje jednoduchost používání spolu s ostatními nástroji od společnosti *Mentor*. *Questa Formal* například obsahuje možnost využít získaný protipříklad a odsimulovat danou situaci v simulátoru pro jednodušší pochopení, co chybu způsobuje. Kromě kontroly tvrzení (*PropCheck*) *Questa Formal* obsahuje i několik automatických nástrojů (*AutoCheck*, *Connectivity check* atd.).

PropCheck [9] podporuje *RTL* modely popsané ve dvou jazycích (*VHDL* a *Verilog*) a popis vlastností ve třech jazycích (*SVA*, *PSL* a *OVL*). Zvolil jsem *SVA* a *Verilog*, aby syntax byla co nejbližší, jelikož *UVM* je implementované v jazyce *SystemVerilog*.

Kapitola 5

Tvrzení v SVA

Pro úvod do způsobu psaní tvrzení v *SVA* jsem využil tutoriál od Duolos [11]. Mnoho informací o dobrých *SVA* praktikách pochází z této publikace [17].

SVA obsahuje dva druhy tvrzení: Okamžité tvrzení a Souběžné tvrzení (*immediate assertion*, *concurent assertion*). Okamžitá tvrzení se využívají v sekvenční části kódu, kde označují podmínku splnění vlastnosti, když se provádění kódu dostane do určité části. Často se využívají v softwarovém programování (např. na kontrolu správných vstupů funkce, správnost mezi výpočtu apod.). Souběžná tvrzení se využívají na specifikaci všeobecné vlastnosti (např. dva výstupy z modulu se nesmí nikdy rovnat).

Tvrzení mohou samozřejmě být použity i v simulaci, ale jelikož velké množství tvrzení zpomalí rychlost simulace, většinou není žádoucí, aby byla použita stejná tvrzení pro formální verifikaci a simulaci. Vzniká tedy potřeba nějakého systému, který oddělí tvrzení kódu. Okamžitá tvrzení musí být uvnitř kódu, proto je tedy oddělení od zdrojového kódu složitější. Naproti tomu souběžná tvrzení lze kompletně oddělit od implementace a pouze je k implementaci připojit, když je to žádoucí.

5.1 Připojení tvrzení k implementaci

Pro připojení souběžných tvrzení potřebujeme vytvořit nový modul, ve kterém specifikujeme tvrzení a použijeme příkaz `bind` jazyka *SystemVerilog*.

```
bind testovanyModul modulSAssertiony nazevInstanceModulu(.vstupA(vstupA));
```

Příkaz *bind* lze použít buď na modul nebo na specifickou instanci. V případě použití na modul dochází s každou další instancí kontrolovaného modulu zároveň k vytvoření nové instance modulu s tvrzeními. Pokud je příkaz použit na specifickou instanci, vytvoří se pouze jedna instance modulu s tvrzeními. Záleží na způsobu, jakým chceme modul kontrolovat, nicméně většinou je žádoucí zkontrolovat všechny instance. Pokud je modul parametrizován, potřebujeme tyto parametry také specifikovat. To můžeme uskutečnit propojením parametrů modulu s tvrzeními s parametry kontrolovaného modulu. Při vytváření modulu s tvrzeními není třeba, aby měl stejné vstupy a výstupy. V případě, že nepoužijeme stejné, potřebujeme specifikovat v závorce na konci příkazu, jakým způsobem mají být připojeny. Při použití stejného rozhraní stačí zadat `(.*)`. Při definici modulu s tvrzeními musíme všechny výstupy označit za vstupy, aby tento nový modul neměnil činnost toho původního.

5.2 Psaní tvrzení

5.2.1 Referenční řešení

Možná se může zdát lákavé danou činnost naimplementovat a pouze uvést tvrzení, že oba výsledky jsou totožné. Tímto postupem se ale činnosti zkomplikují. Pokud by dané tvrzení selhalo, bylo by složitější zjistit příčinu problému. Např. tvrzení pro kontrolu sčítání: Výhodněji se jeví zápis tvrzení, že $A+B=C$, než si $A+B$ spočítat mimo a pouze zkontrolovat výsledek. Pokud tvrzení selže, vidíme jaké jsou vstupy a lze jednoduše ověřit zda je chybný výsledek, nebo tvrzení.

5.2.2 Název

V programování se vyplácí popisovat názvy proměnných co nejdetailněji. To stejné platí i o názvech tvrzení. V případě, že název tvrzení označíme stylem "SpravnyVysledek", pokud tvrzení selže nevíme vlastně, jak má výsledek vypadat. Při detailním popisu sice může mít název desítky znaků, ale pokud tento postup usnadňuje dřívější odhalení chyb, je moudré se znaky nešetřit. Všechna tvrzení v mém kódu jsou napsána stylem:

```
ASSERT_vysledek_eq_A_plus_B_when_scitani.
```

Velké ASSERT na začátku pomůže určit počátek názvu tvrzení v celém jméně jeho instance. Dále následuje signál, který je tvrzením kontrolován, a určení jeho výsledku. Následuje slovo *when*, za nímž je definováno, kdy se tvrzení použije. Tento konkrétní příklad je samozřejmě zjednodušen pro vysvětlení. U tvrzení, které popisují všeobecně známou problematiku (jako sčítání) ničemu neškodí zapsat název tvrzení tímto způsobem:

```
ASSERT_spravny_vysledek_scitani.
```

5.2.3 Syntax

Okamžitá tvrzení se v *SVA* zapisují:

```
nazev_tvrzeni: assert (podminka) prikaz_kdyz_plati;
else prikaz_kdyz_tvrzeni_neplati;
```

Jedinnou povinou část představuje samotný příkaz a podmínka.
Souběžné tvrzení lze zapsat např.:

```
property vysledek_eq_A_plus_B_when_scitani;
    @(posedge(CLK))
        scitani |-> ##1 vysledek == (A+B);
endproperty
ASSERT_vysledek_eq_A_plus_B_when_scitani:
assert property (vysledek_eq_A_plus_B_when_scitani);
```

Nebo lze vlastnost definovat přímo v závorkách u `assert property`.

5.2.4 Definování podmínky použití tvrzení

Pokud chceme definovat, kdy má být výsledek kontrolován, stačí použít implikaci |->. Pro určení, kdy naopak nemá být kontrola provedena, lze využít příkaz `disable iff(podmínka)` po definici závislosti na hodinový signál.

5.2.5 Zpoždění signálu

Pokud má být výsledek známý, za/po několik taktů hodinového signálu, lze kontrolu zpozdit:

```
//Když A je pravdivé B musí být pravdivé:
A |-> ##2 B //za dva takty
A |-> ##[1:3] B//za jeden až tři takty
A |-> B[*2] // po dobu dvou taktů
```

Pokud by po implikaci byl signál zpožděný o jeden takt (|-> ##1), tak lze implikaci zapsat jako |=>. Tento způsob je sice kratší, ale dle mého názoru je méně přehledný, a proto jsem vždy používal první zápis.

5.2.6 Proměnné

Proměnná uvnitř tvrzení může přinášet užitek, pokud chceme kontrolovat zpoždění vstupu apod. Využil jsem ji při ověřování registru, pokud se do registru zapíše hodnota, příští hodinový takt bude na výstupu tato hodnota.

```
property output_eq_previous_input_we;
    logic [bit_width-1:0] prev_d;
    @(posedge CLK) disable iff ( RST == reset_level )
        (WE, prev_d = D) |-> ##1 Q == prev_d;
    endproperty
ASSERT_OUTPUT_EQ_PREVIOUS_INPUT_WE:
assert property (output_eq_previous_input_we);
```

Při zápisu definice tvrzení přímo v závorkách u `assert property` se vyskytly problémy, když jsem chtěl použít proměnnou (nejspíš chyba *PropChecku*). Proto je uvnitř mého kódu použita pouze oddělená definice, aby byl zápis jednotný.

5.2.7 Kontrola souběžného přiřazení

Ačkoliv *SVA* umožňuje kontrolu souběžného přiřazení (příkaz `assign` v jazyce *Verilog*), *PropCheck* tuto možnost nemá a proto je nutné využít jiný způsob. *SystemVerilog* obsahuje `$global_clock`, který se aktivuje s každým krokem formální verifikace. Nastavování všech signálů se během formální verifikace synchronizuje pomocí tohoto signálu, proto je dostačující, když kontrolujeme výsledky s jeho každou hranou.

5.2.8 Pokrytí vlastnosti

Místo tvrzení o pravdivosti vlastnosti lze uvést, že je žádoucí, aby uvedená vlastnost nastala. Syntax zůstává naprosto stejná, jenom místo `assert property (vlastnost)` se použije `cover property (vlastnost)`. Formální nástroj se poté bude snažit navodit situaci uvedenou ve vlastnosti, pokud se to podaří, uvede příklad vstupu, při kterém nastane.

5.2.9 Užitečné SVA funkce

SVA obsahuje několik užitečných funkcí, které analyzují stav signálu během posledních taktů hodinového signálu.

Zjišťování zda signál změnil hodnotu

Funkce `$rose` vrací *True* pokud se od posledního hodinového taktu změnil nejméně důležitý bit z 0 na 1. Funkce `$fell` naopak kontroluje změnu nejméně důležitého bitu z 1 na 0.

Funkce `$stable` vrací *True*, pokud se signál od posledního hodinového signálu nezměnil. V mém kódu jsem tuto funkci využil pro kontrolu, zda se výstupní signál registru nezmění, když není umožněn zápis do registru, a nebo když je pozdrženo provádění instrukce.

Tyto funkce navíc přijímají jako argument hodinový signál, ke kterému se signál vztahuje. Implicitně se využívá hodinový signál, jenž je použit pro řízení dané části kódu (Pokud žádný takový signál neexistuje, použije se výchozí hodinový signál).

Zjištění minulé hodnoty signálu

Pro zjištění minulé hodnoty využil pomocnou proměnnou, nicméně by bylo možné využít i vestavěnou funkci `$past`, která přijímá až čtyři argumenty: Signál, jehož hodnotu chceme zjistit (Jediný povinný argument).

Celé číslo specifikující o kolik hodinových signálů zpět, má být hodnota získána (Implicitně 1).

Stejně jako u předchozích funkcí lze specifikovat hodinový signál, na který je tato hodnota vázána.

Zmíněná funkce navíc umožňuje použít výraz, který pokud neplatí v daném hodinovém signálu, v důsledku vede k ignorování hodinového signálu (*clock gating*).

Analyzování hodnoty

Funkce `$onehot` a `$onehot0` lze použít pro zjištění zda signál obsahuje právě jeden, respektive nejvýše jeden, bit nastavený na 1. `$isunknown` vrací $True$, pokud kterýkoliv bit signálu obsahuje neznámou hodnotu (X, Z). `$countones` spočítá množství bitů nastavených na hodnotu 1.

5.3 Užitečná SVA funkcionalita nepodporovaná v programu PropCheck

PropCheck bohužel neimplementuje celou funkcionalitu *SVA*, takže pokud je nějaká funkcionalita potřeba, vyvstává nutnost situaci vyřešit jiným způsobem. Vždy dostupný způsob řešení spočívá v implementaci funkcionality a ověření výsledku (není sice ideální, ale může být jediný).

SVA obsahuje operátory `until_with` a `s_until_with`, které kontrolují, zda je výraz na levé straně roven 1 do doby, než se výraz na pravé straně nastaví na 1. Rozdíl mezi těmito funkcemi spočívá ve skutečnosti, že `s_until_with` vrátí $True$, pouze pokud nastal okamžik, kdy oba výrazy byly pravdivé ve stejnou chvíli, zatímco u `until_with` se může hodnota na levé straně rovnat 0 o jeden takt dříve, než se nastaví výraz na pravé straně. Skutečnost, že tuto funkcionalitu *PropCheck* nepodporuje, jsem vyřešil pomocí referenčního řešení.

SVA rovněž umožňuje tvrzení, která obsahují více hodinových signálů. Takové tvrzení může být užitečné, pokud by náš modul obsahoval více hodinových signálů. Ale toto tvrzení lze řešit např. za využití `$global_clock`.

5.4 Nevýhody nástroje PropCheck

Nevýhodu *PropChecku* (a jiných programů od společnosti *Mentor*) vidím v řešení chybných a nepodporovaných zápisů. Vypíší varování a tento zápis ignorují místo toho, aby vyvolaly chybu při kompilaci. Pokud je tedy použita nepodporovaná funkcionalita v nějakém tvrzení, potom toto tvrzení není zkompilováno. Pokud design obsahuje i jen desítky tvrzení, může se stát, že si nevšimnete, že jedno chybí. V případě napojení signálu na nějaký modul, který není definovaný, také nedojde k chybě překladu. Pokud o této skutečnosti nevíte, může být poměrně špatně odhalitelná chyba, kterou to způsobí.

5.5 Kontrolování výsledků

5.5.1 Potvrzené tvrzení

Potvrzení všech tvrzení se jeví samozřejmě jako nejideálnější, ale neznamená, že design neobsahuje chybu. Např. může být chyba v tvrzení, nebo tvrzení nepopisují všechny možnosti.

5.5.2 Prázdné potvrzení

Prázdné potvrzení (*Vacuous proof*) nastává, když při použití implikace, kdy první část nikdy nenastane. Tvrzení je sice pravdivé, protože když je první část vždy 0 výsledek je vždy roven 1, ale zároveň toto tvrzení nic neprokáže. Prázdné potvrzení nemusí představovat problém, pokud se používá jeden modul na více místech a na nějakém z těchto míst není vyžadována plná funkcionalita. Ale zároveň může indikovat chybu, když se objeví v modulu ve kterém se objevit nemá.

5.5.3 Selhávající tvrzení

Pokud tvrzení selže, musíme zjistit chybné místo. Nejdříve by měla být zkontrolována, správnost daného tvrzení, protože kontrola je pravděpodobně jednodušší než hledání chyby v designu. V tomto případě se potvrzuje užitečnost řádného zapsání názvu tvrzení a používání signálů se kterými se reálně pracuje protože to pomůže odhalit chyby jednodušeji.

Selhávající tvrzení s varováním

Tato chyba znamená, že protipříklad se podařilo získat jiným způsobem než ze vstupu, a proto bývá jednodušší odhalit její původ. Varování bývá způsobeno například nepřipojeným signálem nebo neinicializovaným registrem. V informacích o chybě lze přímo zjistit, jaký signál mimo vstupního byl použit. V mém kódu tato chyba nastala vždy, když jsem zapomněl změnit výstupy modulu na vstupy, protože poté byla hodnota výstupního signálu získávána z modulu s tvrzeními (který samozřejmě nebyl připojený).

5.5.4 Neprokázané tvrzení

Neprokázané tvrzení nastane v tu chvíli, kdy formální běh skončí dříve než se podaří najít důkaz správnosti nebo protipříklad. Následně je možné postupovat dvěma způsoby: Zvýšit délku formálního běhu, nebo zjednodušit tvrzení (popř. ověřit tvrzení jiným způsobem – třeba simulací). Informace o stavu indikované formální analýzou, které značí hloubku, v které se formální analýza nachází mohou pomoci udělat správné rozhodnutí.

5.5.5 Nepokrytelné vlastnosti

Pokud je nějaká vlastnost nepokrytelná, je nutné nejdříve ověřit možnost, vzniku dané situace. Pokud má být situace možná, vzniká potřeba zjištění proč nenastala.

5.5.6 Pokryté vlastnosti

Pokud se vlastnost pokryje, můžeme zjistit příklad vstupu, který to umožní.

Kapitola 6

Implementace

6.1 Kontrolovaný procesor

Procesor, který jsem využil na implementační část této práce, byl vyvinut brněnskou firmou *Codasip*. Zmíněná společnost se zabývá vývojem *IDE* nástrojů na implementaci procesorů a zároveň tyto nástroje využívá pro vývoj svých vlastních procesorů založených na *RISC-V ISA*. Hlavní přednost firmy *Codasip* představuje skutečnost, že vyvíjené procesory, jsou vysoce přizpůsobitelné. Codasip nabízí procesory implementované v několika jazycích na popis hardwaru a možnosti použití kombinace různých rozšíření. Procesor lze tedy uzpůsobit konkrétním požadavkům.

Konkrétně jsem využil procesor *BK3*, který implementuje zřetězené zpracování o třech částech se zpracováním instrukcí v pořadí, v jakém jsou načítány z paměti. Zvolený procesor implementuje 32 bitovou základní instrukční sadu, spolu s rozšířeními: *Zicsr*, *Zifencei*, *User Mode* a obsahuje kontrolér pro práci s externími přerušeními. *Zicsr* představuje standardní rozšíření definující 6 instrukcí pro práci s kontrolními registry. *Zifencei* je název standardního rozšíření definující instrukci `fence.i`, která vyčistí procesor od zpracovávaných instrukcí a zapíše všechna data z mezipaměti do paměti. Procesor obsahuje navíc dva nestandardní kontrolní registry používané pro práci s externími přerušeními.

Ke kontrole procesoru jsem přistoupil až v pozdější fázi vývoje, kdy už byl ve značné míře ověřen pomocí funkční verifikace a některých automatických formálních nástrojů. Neočekával jsem proto, že objevím příliš mnoho chyb, ale spíše jsem předpokládal, že budu potvrzovat správnost.

6.2 Postup při implementaci

6.2.1 Spouštěcí skript

Formální běh je spouštěn skriptem, který používá Questa Formal ve třech fázích. Tyto fáze jsou popsány v *tcl* skriptech (součástí instalace *Questa Formal* je i tutoriál pro *PropCheck*, z něhož jsem *tcl* skripty upravil). Pokud nějaká část selže, další se nespustí. První dvě části se spouštějí v režimu příkazové řádky, třetí část se zobrazí graficky. Nejdříve se vyčistí pracovní složka a načtou všechny zdrojové soubory. Načítané soubory jsou popsány ve dvou souborech, které obsahují seznamy se zdrojovými *RTL/Assertion* soubory. V druhé části se spustí kompilace, pokud proběhne v pořádku, spustí se *PropCheck*. Zápisy z těchto běhů jsou ukládány do oddělených souborů. Ve třetí části se zobrazí výsledky.

6.2.2 Registry

Jako první jsem se rozhodl začít popisovat vlastnosti registrů, které jsou použity v rámci procesoru, protože je velmi malá pravděpodobnost, že obsahují chybu (jsou použity často, proto by se jejich chyba v simulaci pravděpodobně projevila). Díky tomuto rozhodnutí jsem měl prostor pro zkoumání, jakým způsobem pracovat s *PropCheckem* a *SVA*.

Konkrétně jsem kontroloval, zda registr obsahuje správná data, když jsou do něj zapsána. Když je registr vyčištěn, že obsahuje výchozí hodnotu. Když má být hodnota v registru pozdržena, že se doopravdy nezmění. A když registr není zapisován ani mazán, že zůstane hodnota nezměněna.

6.2.3 Dekodér

Abych ověřil, že jsou instrukce prováděny způsobem, jakým je definováno, začal jsem od prvního bodu, kde se začne instrukce zpracovávat – dekodér. Výhodou také bylo, že dekodér obsahuje jediný vstup (instrukci) a tudíž jsem nemusel zkoumat práci se vstupními řídícími signály. Většina informací potřebných k dekodéru byla získatelná z *RISC-V* specifikace [1], nicméně v případě některých signálů jsem musel zkoumat, jakým způsobem se nastaví a později kontrolovat způsob jejich zpracování (např. řídící signál pro ALU).

Tvrzení o instrukcí jsem shlukoval podle operačních kódů (nejnižších 7 bitů). Instrukce, které mají stejný operační kód, nastavují většinu signálů stejným způsobem. U každého z operačních kódů jsem kontroloval, že signály, s kterými žádná z instrukcí nepracuje, jsou nastaveny na 0 (např. že výpočetní instrukce nenastavují signál značící skok). Kontroluje se, zda každý operační kód nastavuje správně signál značící chybnou instrukci (například většina výpočetních instrukcí pracující s dvěma registry musí mít 7 nejvyšších bitů nastavených na 0). Dále kontroluji, zda každá instrukce nastavuje správné zdrojové operandy (některé instrukce pracují se dvěma registry, jiné s registrem a okamžitou hodnotou apod.). Pokud instrukce pracuje s okamžitou hodnotou, tak je nastaven správný typ, jakým způsobem jsou okamžité hodnoty zakódovány v instrukci. Instrukce pracující s pamětí nastavují správně, zda se čte nebo zapisuje a s jakou velikostí dat se pracuje. Jakým způsobem se nastavují řídící signály pro ALU a modul s kontrolními registry. Skokové instrukce nastavují signál značící skok. Pokud instrukce neobsahuje legální operační kód, tak je nastaven signál značící nelegální instrukci.

6.2.4 Provádění instrukcí

Poté jsem popisoval jednotlivé moduly, které provádějí instrukce (ALU, *komparátor*, komunikace s pamětí). Z dekodéru jsem získal informace o způsobu, jakým instrukce nastavují jednotlivé datové a řídící signály pro některé moduly. Ověřoval jsem, že chování jednotlivých modulů odpovídá dekódovaným instrukcím. Díky výstupním signálům z jednotlivých modulů jsem mohl určit některé řídící signály (např. signál o pozdržení z důvodu přístupu k paměti).

Pro výsledek jednotlivých operací jsem psal tvrzení, zda správně zpracují data, nicméně jsem nepsal pro každý typ dat zvláštní tvrzení. Pro data jsem použil pomocný signál, který se nastavoval v závislosti na použitých hodnotách (například konkrétní typy okamžitých hodnot) a výsledky jsem kontroloval s pomocí těchto pomocných signálů. Také jsem kontroloval, zda tyto moduly mají na výstupu 0, když nezpracovávají žádné operace. Některé instrukce je možné provádět pouze v *machine* módu (*mret*, *wfi*), tudíž jsem také kontroloval zda vyvolají výjimku v uživatelském módu.

6.2.5 Modul s kontrolními registry

Popis modulu pro kontrolní registry představoval nejnáročnější část ze všech modulů, protože kromě explicitního zápisu je většina registrů nastavována z různých důvodů (výjimky, přerušení apod.) a zároveň zpracovává výjimky a přerušení. Každý z registrů má vlastní pravidla o způsobu nastavování, a jakých hodnot může nabývat. Nicméně některé registry nemusí být implementovány (musí existovat, ale obsahují hodnotu 0). Informace o způsobu fungování standardních registrů byly získány z privilegované specifikace [2]. Informace o nestandardních registrech jsem využil z katalogového listu (*datasheet*).

Kontroloval jsem, zda jsou na výstupu data ze správného registru, když se čte. Registry, které jsou pouze ke čtení nesmí umožnit zápis. Instrukce, které mají pouze přečíst data z registru, do něj nezapisují (např. instrukce CSRRS se zdrojovým registrem x0). Registry obsahují pouze data která mohou obsahovat.

Dále jsem kontroloval všechny možnosti, za jakých podmínek je v kontrolních registrech měněná hodnota. Hlavně jsem se tedy zaměřil na části zpracovávající výjimky a přerušení. Výjimky, které mohou nastat (např. nevalidní instrukce, skok na nezarovnanou adresu, zápis do neexistujícího kontrolního registru) jsou zpracovány a jednotlivé chyby jsou rozlišeny správně (správný důvod v registru *mcause*, skok na správnou adresu apod.). Kontroloval jsem reakci na jednotlivá přerušení (vnější, časové, softwarové) a zda jsou určeny správně.

6.2.6 Řízení zřetězeného zpracování

Prostřednictvím informací o způsobu, kterým zpracovávají data jednotlivé moduly, a podle jimi nastavovaných řídících signálů, jsem mohl zkontrolovat způsob řízení zřetězeného zpracování.

Popisoval jsem, jakým způsobem jsou předávány data do jednotlivých funkčních bloků. Jakým způsobem se zdržuje provádění určitých částí (např. z důvodu čekání na paměť). Zda jsou načítány instrukce ze správných adres. Také jsem kontroloval, zda do registru *x0* nelze zapsat jiná hodnota než 0.

6.3 Odhalené chyby a nedostatky

Výskyt chyb naplnil mé očekávání, objevil jsem jich poměrně málo a v komponentách, u kterých jsem to považoval za nejpravděpodobnější.

Dekodér sice dekóduje všechny instrukce, ale během jeho programování se zapomnělo na speciální variantu instrukce `FENCE`. Tato instrukce se používá pro synchronizaci vstupů a výstupů procesoru a vrchní 4 bity jsou nastaveny na hodnotu 0. Výjimkou je `FENCE.TSO`, s nastavenými vrchními 4 bity na hodnotu 1000.

Pro kontrolu nestandardních kontrolních registrů jsem využíval informace z katalogového listu, což vedlo k odhalení chyb, které sxe v něm nacházely. Procesor obsahuje dva nestandardní kontrolní registry pro ovládání přerušení. V katalogovém listu měly tyto registry zaměněné adresy. Zmíněná chyba je sice teoreticky odhalitelná v simulaci, ale jelikož cílový registr bývá v assembleru identifikován jeho názvem, nikoli jeho adresou, jedná se o chybu složitější na odhalení.

Procesor také nemá úplně správnou reakci na přerušení. Pokud je příznak přerušení nastaven než je zpracováno, vše funguje jak má. Ale pokud nastaven nezůstane, je přerušení vždy určeno jako časové přerušení, což způsobí špatná data v *mcause* registru (špatný důvod). Může rovněž způsobit skok na špatnou adresu (záleží na hodnotě v *mtvec* registru).

Dále jsem objevil že signál, který zdržuje části procesoru z důvodu čtení po zápisu by bylo možné optimalizovat. Tento signál je v procesoru nastaven, když se shoduje adresa druhého zdrojového registru s předchozím zapisovaným i když není v rámci instrukce druhý registr použit. Tato skutečnost sice nezpůsobí chybu, ale může způsobit zbytečné zdržení instrukce, když náhodou tato situace nastane. Ačkoliv se tento nedostatek v simulaci projeví jeho odhalení téměř není reálné, protože by to bylo možné pouze, kdyby se kontrolovalo časování instrukcí a nebo náhodou, při sledování jednotlivých signálů z jiného důvodu.

Navíc *PropCheck* ještě automaticky odhalil několik zbytečných částí kódu.

Kapitola 7

Závěr

RISC-V je zajímavá architektura, která umožňuje procesory přizpůsobit širokému spektru přístrojů. Nicméně během tvorby takového procesoru se může objevit mnoho chyb, které musí být odhaleny co nejdříve. Funkční verifikace se snaží zajistit, že navržený hardware funguje podle specifikace, Jelikož verifikace představuje důležitý a hlavně zdlouhavý proces návrhu hardwaru, je snaha co nejvíce tento proces zjednodušit a vylepšit. Z tohoto důvodu jsou vyvíjeny různé postupy, jakým způsobem verifikovat.

Jeden z takovýchto postupů představuje *UVM*, jejíž hlavní myšlenka spočívá ve znovupoužitelnosti verifikačních prostředí. *UVM* obsahuje mnoho způsobů, jakým je možné měnit činnost verifikačního prostředí. Nicméně *UVM* se zaměřuje na testování v podobě simulace a potřebuje, aby verifikační prostředí generovalo vstupní data, která budou testovat daný hardware. Nevýhoda tohoto přístupu spočívá v nemožnosti pokrytí všech možných vstupů. A tudíž je nutné využívat i jiné postupy.

Formální verifikace představuje jiný přístup oproti funkční verifikaci. Formální verifikace využívá matematiky k ověření správnosti, zatímco funkční verifikace ověřuje správnost modelu za použití simulace. Hlavní přínos ve využití formálního nástroje pro kontrolu tvrzení shledávám v kontrolování dekodéru, kontrolních registrů, *ALU* a dalších modulů - obzvláště těch, které obsahují velké množství možných vstupů.

Správné dekódování instrukce je složitější na ověřování v simulaci, protože instrukce mohou nabývat velkého množství hodnot. Proto je v podstatě nemožné ověření, zda všechny nevalidní instrukce doopravdy způsobí chybu a všechny validní jsou správně dekódovány. Podobné důvody lze najít u kontrolních registrů. Ověřování, zda procesor obsahuje pouze registry, které má obsahovat, nebo zda lze do registru zapsat pouze ve správném módu, je pomocí simulace poměrně zdlouhavé, a pokud se k tomu ještě přidá ověření zda registr obsahuje vždy validní hodnotu, potom je v podstatě nemožné toto v simulaci zkontrolovat. Oproti tomu formální nástroj tuto funkcionalitu zkontroluje během několika desítek vteřin. Pro ideální použití tvrzení, je vhodná řádná specifikace chování jednotlivých modulů. Ověřování způsobu, jakým se řídí procesor, je pomocí tvrzení složité, protože může jednoduše vzniknout chyba v samotném tvrzení. K ověřování způsobu řízení procesoru se spíše hodí využít simulaci a kontrolovat správnou činnost s využitím náhodných dat, zpožďováním odpovědí paměti a zkoušet přerušení v co nejvíce situacích.

Cílem této práce bylo zjistit, jakým způsobem funguje funkční verifikace a konkrétně *UVM*. Dalším cílem bylo prozkoumat, jakým způsobem do funkční verifikace zapadá formální verifikace, jaké existují nástroje a metody práce s nimi. A nakonec využít formální verifikaci tvrzení k ověření procesoru s architekturou RISC-V a odhalit chyby, které se ne-

podařilo odhalit s pomocí jiných způsobů. Odhalené chyby se objevily v komponentách, které je složitější kontrolovat pomocí simulace, což potvrdilo moje očekávání.

Literatura

[1] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019.

[2] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, June 2019.

[3] Fisher J.A., Faraboschi P., Young C. (2011) VLIW Processors. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA.

[4] Mehta, A. B.: ASIC/SoC Functional Design Verification. Springer International Publishing,2018, ISBN: 9783319866208.

[5] *Codasip Studio | Codasip* [online]. [cit. 2020-5-31]. Dostupné z: https://codasip.com/codasip-studio/.

[6] *JasperGold Formal Verification Platform (Apps)* [online]. [cit. 2020-5-24]. Dostupné z: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[7] *Questa Formal Verification* [online]. [cit. 2020-5-24]. Dostupné z: https://www.synopsys.com/verification/static-and-formal-verification.html.

[8] *Questa Formal Verification - Mentor Graphics* [online]. [cit. 2020-5-24]. Dostupné z: https://www.mentor.com/products/fv/questa-formal/.

[9] *Questa® Property Checking - Mentor Graphics* [online]. [cit. 2020-5-24]. Dostupné z: https://www.mentor.com/products/fv/questa-property-checking.

[10] *RISC-V - Debian Wiki* [online]. [cit. 2020-5-24]. Dostupné z: https://wiki.debian.org/RISC-V.

[11] *SystemVerilog Assertions Tutorial* [online]. [cit. 2020-5-24]. Dostupné z: https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/.

[12] *UVM - The Universal Verificaton Methodology* [online]. [cit. 2020-5-24]. Dostupné z: https://www.doulos.com/knowhow/sysverilog/uvm/.

[13] *UVM Tutorial for Beginners* [online]. [cit. 2020-5-24]. Dostupné z: https://www.chipverify.com/uvm/uvm-tutorial.

[14] *Verification Academy - The most comprehensive resource for verification training. | Verification Academy* [online]. [cit. 2020-5-24]. Dostupné z: https://verificationacademy.com/.

[15] Allan, G., Chidolue, G., Ellis, T., Foster, H., Horn, M. et al. *Coverage Cookbook* [online]. 2019 [cit. 2020-5-24]. Dostupné z: https://verificationacademy.com/cookbook/coverage/pdf.

[16] Araújo, P. *UVM Guide for Beginners - Pedro Araújo* [online]. [cit. 2020-5-24]. Dostupné z: https://colorlesscube.com/uvm-guide-for-beginners/.

[17] Cummings, C. E. *SystemVerilogAssertions - BindfilesBestKnownPracticesforSimpleSVAUsage* [online]. Silicon Valley, CA: Sunburst Design, Inc., duben 2016 [cit. 2020-5-24]. Dostupné z: http://www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf.

[18] Goos, G., Hartmanis, J. et al. *Exploitation of fine-grain parallelism*. Springer Science & Business Media, 1995.

[19] Guzel, B. *Top 15+ Best Practices for Writing Super Readable Code* [online]. Březen 2011 [cit. 2020-5-30]. Dostupné z: https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118.

[20] MOLINA, A. a Cadenas, O. Functional verification: Approaches and challenges. *Latin American applied research Pesquisa aplicada latino americana = Investigación aplicada latinoamericana*. Leden 2007, roč. 37.

[21] Morley, D. a Parker, C. S. *Understanding computers: Today and tomorrow, comprehensive*. Cengage Learning, 2014.

[22] Olivka, P. *Procesory CISC a RISC* [online]. 2010 [cit. 2020-06-01]. Dostupné z: http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf.

[23] Rajendra, P. *Basics of RTL Design and Synthesis* [online]. Silicon Valley, CA: [b.n.], únor 2018 [cit. 2020-6-1]. Dostupné z: http://smdpc2sd.gov.in/downloads/IEP/IEP%208/24-02-18%20Rejender%20pratap.pdf.