VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVĚ AKCELEROVANÁ FUNKČNÍ VERIFIKACE PROCESORU

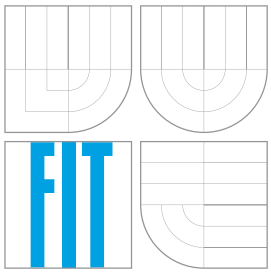
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN FUNIAK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVĚ AKCELEROVANÁ FUNKČNÍ VERIFIKACE PROCESORU

HARDWARE ACCELERATED FUNCTIONAL VERIFICATION OF PROCESSOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN FUNIAK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARCELA ŠIMKOVÁ

BRNO 2013

Abstrakt

Mezi aktuálně používané verifikační přístupy patří funkční verifikace. Při funkční verifikaci se ověřuje korektnost implementace počítačového systému vzhledem k specifikaci. Slabým místem v rámci přístupu funkční verifikace je její časová náročnost, na kterou má vliv pomalá softwarová simulace implicitně paralelních hardwarových systémů. V této práci je představeno řešení využívající hardwarovou akceleraci funkční verifikace procesoru. Úvodní kapitoly tvoří teoretický základ pro následující kapitoly, ve kterých se nachází analýza a výběr řešení, návrh verifikačního prostředí a implementační detaily. Závěr práce obsahuje testování výsledného produktu, zhodnocení výsledků práce a vyhlídky do budoucna.

Abstract

Functional verification belongs among the current verification approaches. Functional verification checks the correctness of the implementation of the system, due to its specification. The weakness of the functional verification approach is time consumption caused by slow software simulation of implicitly parallel hardware systems. This paper presents a solution for using a hardware accelerated functional verification of the processor. The introductory chapters form the theoretical basis for the following chapters, that include a choice of solutions, an analysis, a design of a verification environment and implementation details. The conclusion includes tests of the final product, evaluation of the results and the future work perspectives.

Klíčová slova

hardwarová akcelerace, funkční verifikace, verifikace procesoru, SystemVerilog, FPGA, Codasip

Keywords

hardware acceleration, functional verification, verification of processor, SystemVerilog, FPGA, Codasip

Citace

Martin Funiak: Hardwarově akcelerovaná funkční
verifikace procesoru, bakalářská práce, Brno, FIT VUT v Brně, 2013

Hardwarově akcelerovaná funkční verifikace procesoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Marcely Šimkové. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Funiak
13. května 2013

Poděkování

Zde bych se rád poděkoval vedoucí mé bakalářské práce Ing. Marcele Šimkové za množství užitečných rad a také za čas, usilí a trpělivost, které byly za potřebí pri konzultacích této práce. Rád bych poděkoval také Ing. Josefu Potešilovi za poskytnuté konzultace a brzké odpovědi při mailové komunikaci.

Obsah

Kapitola 1

Úvod

V dnešnej dobe sa počítačové systémy nachádzajú všade okolo nás a sú neoddeliteľnou súčasťou nášho každodenného života. Tieto systémy sa aplikujú na rozsiahlu škálu úloh od menej dôležitých až po tie kriticky dôležité vyžadujúce odozvu systému v reálnom čase, od ktorých závisia ľudské životy. Používatelia počítačových systémov sa spoliehajú na to, že tieto systémy fungujú korektne a neobsahujú chyby.

Vedná disciplína, ktorá v rámci oboru informačných technológii overuje korektnosť systému vzhľadom na danú špecifikáciu sa nazýva verifikácia.

Pre verifikáciu počítačových systémov sa používajú rôzne verifikačné prístupy: simulácia, funkčná verifikácia, formálna verifikácia a iné zamerané na testovanie obvodu po výrobe. Podrobnejší popis verifikačných prístupov sa nachádza v kapitole 2.

Táto práca je zameraná na funkčnú verifikáciu. Ide o prístup, ktorý porovnáva implementáciu obvodu na RTL úrovni s jeho špecifikáciou a to všetko ešte pred začatím výroby. Výhodou funkčnej verifikácie je prevencia chýb v počiatočných fázach návrhu a tiež vysoká miera abstrakcie, preto je možné testovať aj komplexné systémy. Práva vďaka týmto výhodám sa funkčná verifikácia stáva čoraz populárnejšou. Nevýhodou funkčnej verifikácie je jej časová náročnosť, ktorá je podmienená využívaním pomalej softvérovej simulácie implicitne paralelných hardvérových systémov. Naviac, s rastúcou komplexnosťou verifikovaných obvodov exponenciálne rastie celková doba verifikácie. To je jeden z dôvodov, prečo proces verifikácie a testovania podľa [8] tvorí až 70% času pri vývoji počítačového systému.

Pre urýchlenie pomalého behu funkčnej verifikácie má zmysel použiť hardvérovú akceleráciu. Práve ona presúva časť verifikačného prostredia obsahujúcu verifikovanú jednotku do hardvéru.

Naviac, s príchodom programovacích jazykov určených pre verifikáciu, ako napríklad SystemVerilog, ε alebo OpenVera a štandardných verifikačných metodík ako sú OVM (angl. *Open Verification Methodology*) alebo UVM (angl. *Univerzal Verification Methodology*) sa umožnil jednoduchší, efektívnejší a tiež rýchlejší návrh samotných verifikačných prostredí. Popis týchto metodík, ako aj rôznych metód funkčnej verifikácie zahŕňa kapitola 3. Je potrebné podotknúť, že vďaka verifikačným metodikám sa zjednodušil aj vývoj komponentov verifikačných prostredí pre hardvérovú akceleráciu, ktoré sú využité v tejto práci.

Jadrom práce je hardvérová akcelerácia funkčnej verifikácie procesora Codix, ktorý je navrhnutý pomocou systému Codasip. Systém Codasip ponúka integrované vývojové prostredie, ktoré umožňuje rýchle vytváranie aplikačne špecifických inštrukčných procesorov (angl. *ASIP - Application Specific Instruction-set Processor*) a multiprocesorových systémov na čipe (angl. *MPSoC - Multiprocessor Systems on a Chip*). Podrobnejší popis systému Codasip a procesora Codix je uvedený v kapitole 4. Kapitola 5 predstavuje existujúce rieše-

nia zamerané na akceleráciu funkčnej verifikácie a zdôvodňuje, prečo bolo výhodné použiť v tejto práci voľne dostupnú akceleračnú platformu HAVEN. Kapitola 6 sa zaoberá návrhom verifikačného prostredia pre procesor Codix. Na kapitolu týkajúcu sa návrhu nadväzuje kapitola 7 obsahujúca implementačné detaily. V kapitole 8 je popísané testovanie výsledku práce. Zhodnotenie práce a vyhliadky do budúcnosti obsahuje kapitola 9.

Kapitola 2

Verifikačné prístupy

Verifikácia je potrebná preto, aby sme mohli zaručiť, že počítačové systémy plnia požadovanú funkciu a neobsahujú chyby. Cieľom verifikácie je overiť správanie sa verifikovaného systému vzhľadom na zadanú špecifikáciu.

Existujú rôzne prístupy určené pre verifikáciu počítačových systémov. Je možné ich rozdeliť podľa toho, či používajú statické alebo dynamické metódy pre verifikáciu. Statické metódy skúmajú reprezentáciu navrhnutého systému v podobe zdrojového kódu. Nie je potrebná simulácia tohoto systému, dochádza k jeho statickej analýze. Naopak pri dynamických metódach sa pozoruje správanie navrhnutého systému pri množine vstupných hodnôt v simulácii.

Zatiaľ čo v minulosti sa uprednostňovali statické metódy, dnes sa používajú aj dynamické, ako napríklad generovanie náhodných stimulov, simulácia alebo kontrola modelov. V nasledujúcich podkapitolách budú rozobrané základné verifikačné prístupy, ktoré sú aktuálne používané pre verifikáciu hardvérových systémov.

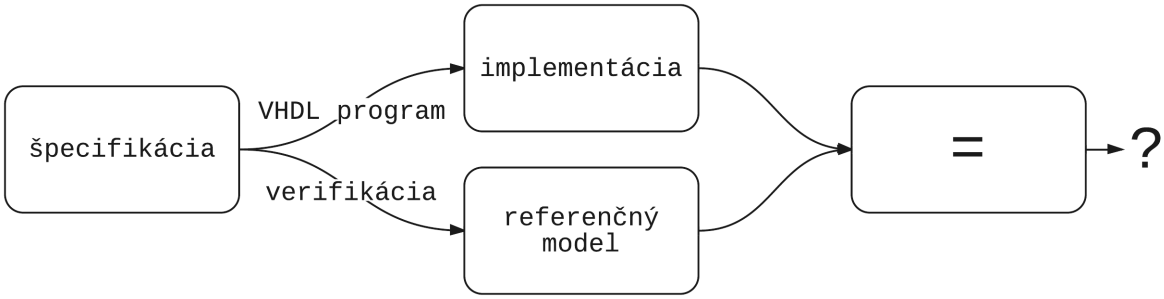
2.1 Formálna verifikácia

Formálna verifikácia umožňuje dokázať správnosť alebo nesprávnosť systému pomocou využitia formálnych metód a abstraktného matematického modelu overovaného systému. Toto je možné považovať za jej nespornú výhodu oproti ostatným verifikačným prístupom, ktoré neumožňujú dokázať správnosť overovaného modelu. Nevýhodou je však vysoká náročnosť na výpočtové prostriedky. Príkladom je problém tzv. stavovej explózie (angl. *State Space Explosion*), kedy dochádza k extrémnemu nárastu počtu kontrolovaných stavov pri prehľadávaní stavového priestoru systému. Taktiež zostrojenie abstraktného matematického modelu, ktorý bude ekvivalentný reálnemu overovanému systému nemusí byť vždy možné.

Verifikačnými metódami, používanými v rámci formálnej verifikácie sú dokazovanie teorémov (angl. *theorem prooving*), statická analýza (angl. *static analysis*) alebo kontrola modelov (angl. *model checking*).

2.2 Funkčná verifikácia

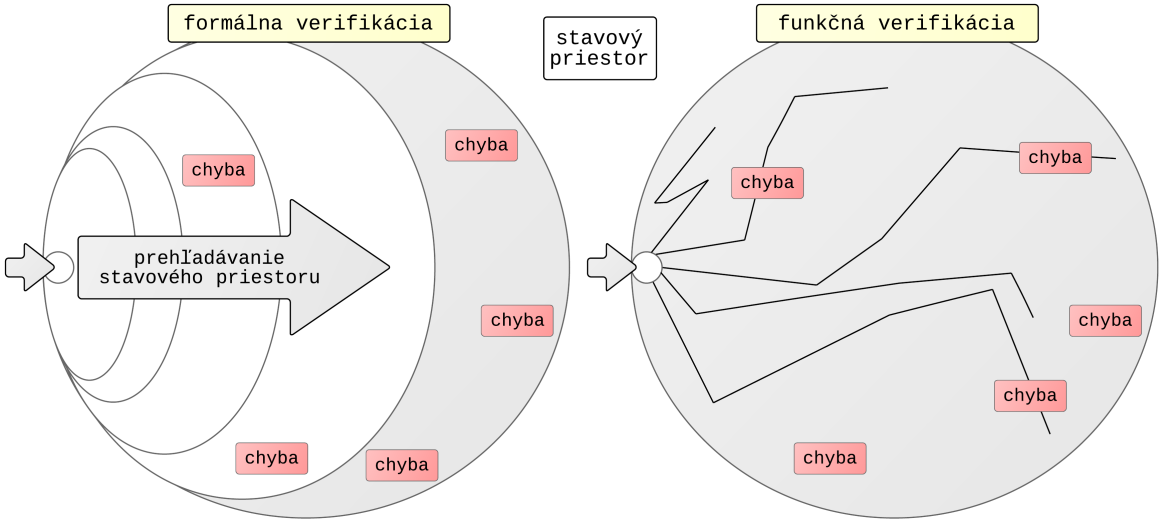
Cieľom funkčnej verifikácie je podľa [8] ukázať, že implementácia počítačového obvodu bude fungovať tak, ako je to zaznamenané v jeho špecifikácii. Schému prístupu funkčnej verifikácie zobrazuje obrázok 2.1. Návrh počítačového obvodu je tvorený na základe slovnej špecifikácie. Slovná reprezentácia informácie môže zahŕňať nejednoznačnosti, protichodné tvrdenia



Obrázok 2.1: Schéma prístupu funčknej verifikácie

a je možné vyložiť ju viacerými spôsobmi. Preto pri návrhu počítačového systému môže dôjsť k neobjektívnej interpretácii špecifikácie a navrhnutý obvod môže obsahovať aj chyby plynúce z nepochopenia alebo zanedbania detailov špecifikácie.

Funkčná verifikácia používa simuláciu ako prostriedok pre overenie správnosti systému, a naviac využíva aj iné pokročilé techniky, ako napríklad generovanie náhodných stimulov, samokotrolné mechanizmy, kontrolu formálnych tvrdení či verifikáciu riadenú úrovňou pokrytia systému. Tieto techniky spolu s využítim programovacieho jazyka SystemVerilog pre funkčnú verifikáciu sú popísané v nasledujúcej kapitole.



Obrázok 2.2: Rozdiel medzi funkčnou a formálnou verifikáciou

Na obrázku 2.2 je možné vidieť základný rozdiel medzi funkčnou a formálnou verifikáciou. Funkčná verifikácia prehľadáva stavový priestor verifikovaného systému pomocou testovacích vektorov do hĺbky, zataľ čo formálna verifikácia prehľadáva stavový priestor systematicky. Ale práve kvôli problému stavovej explózie sa môže stať, že určitá oblasť overovaného systému nebude preskúmaná v dôsledku nedostatku výpočtových prostriedkov.

Kapitola 3

Funkčná verifikácia v SystemVerilog

Táto kapitola popisuje dôležité teoretické fakty týkajúce sa funkčnej verifikácie. V úvodných podkapitolách sú popísané základné pojmy, ako priebeh verifikačného procesu, úroveň verifikácie alebo zloženie verifikačného prostredia. Nasledujúce podkapitoly prechádzajú od popisu všeobecných pojmov k popisu konkrétnych metód funkčnej verifikácie. Zaoberajú sa tvorbou vstupov, kontrolou výstupov a vlastnosťami programovacieho jazyka SystemVerilog. Záverečná podkapitola je venovaná verifikačnej metodike OVM, ktorej využitie vedie k vytvoreniu znovupoužiteľného prostredia pre verifikáciu.

3.1 Verifikačný proces

Hlavnou náplňou verifikačného procesu nie je iba hľadanie chýb, ale zaoberá sa aj overovaním faktu, že implementácia obvodu spĺňa požiadavky definované v špecifikácii [10]. Z tejto špecifikácie vychádza verifikátor, teda člen vývojového tímu, ktorý sa podieľa na verifikácii. Jeho úlohou je zistiť, či verifikovaný systém správne plní svoju funkciu a vyhovuje zadanej špecifikácii. Pre verifikáciu systému sa používajú náhodné testy, ktoré zabezpečia jeho pokrytie. Naviac je potrebné otestovať okrajové podmienky a tiež reakcie systému na chyby pomocou tzv. mutačného testovania. Chyby, ktoré sú odhalené, dokazujú nesúlad medzi navrhnutým systémom a jeho špecifikáciou.

Proces verifikácie by mal prebiehať súbežne s návrhom systému. Návrhár musí pochopiť vstup systému, transformačnú funkcii, ktorú systém vykonáva a formát jeho výstupu preto, aby mohol správne systém navrhnúť. Verifikátor musí takisto rozumieť vstupom, výstupom a transformačnej funkcii systému, no nemal by poznať implementáciu systému.

Návrhár a verifikátor by mali byť dve rôzne osoby, kvôli rozdielnemu pohľadu na špecifikáciu systému a nezávislej implementácii jeho funkčnosti.

3.1.1 Úrovne verifikácie

Vo verifikačnom procese sa pri simulácii overovaného systému zvyčajne postupuje od jednoduchých testov jednotlivých blokov ku testom prepojení blokov [8]. Pri postupnom zvyšovaní abstrakcie sa objavujú chyby súvisiace s prepojením blokov, časovaním a synchronizáciou. Na najvyššej úrovni testujeme celý systém, čo však má výrazný vplyv na rýchlosť simulácie.

Tieto princípy sú vysvetlené v nasledujúcom texte na príklade verifikovaného procesora.

- **Bloková úroveň.** Jedná sa o najjednoduchší postup verifikácie s použitím priamych testov. Na blokovej úrovni sa procesor skladá z jednotiek, ako sú kontrolná jednotka (angl. *Control Unit*), aritmeticko-logická jednotka (angl. *Arithmetic-Logic Unit*) a sada registrov (angl. *Register File*), ktoré môžeme verifikovať osobitne, čím je možné ušetriť čas a výpočtové zdroje. Niektoré chyby sa však objavia až pri spájaní blokov, preto je potrebné verifikovať procesor aj na vyššej úrovni.

- **Spojenia medzi blokmi.** Testuje sa blok a jeho prepojenie so susednými blokmi. Jednotlivé bloky môžu navrhovať rôzni návrhári hardvéru a preto je možné, že pochopia špecifikáciu rozhrania bloku rôzne. Testovanie prebieha pomocou generovania stimulov zo susediacich blokov alebo pomocou formálnych tvrdení (angl. *Assertions*). Výhodou simulácie na nižšej úrovni je jej vysoká rýchlosť. Nevýhodou je to, že chyby sa objavujú postupne pri spájaní blokov.

  Spájanie blokov prináša možnosť vyvolávať stimuly jednotlivými blokmi navzájom. Tým sa zmenšuje vynaložené úsilie pri manuálnom generovaní signálov, avšak za cenu pomalšieho behu simulácie. Verifikáciou blokov procesora je možné otestovať napríklad komunikáciu jednotiek s pamäťou alebo registrovým poľom.

- **Celý systém.** Pri simulácii celého systému výrazne klesá výkon simulácie. Pri testovaní na tejto úrovni je dôležité paralelne simulovať činnosť všetkých blokov. Pri takejto simulácii sa často objavia chyby časovania signálov na rozhraniach jednotlivých blokov. Testovanie prebieha pomocou automaticky generovaných transakcií.

  Verifikácia procesora ako celého systému je náročná na simuláciu ale umožňuje v plnej miere využiť tzv. black-box testovanie, kedy poznáme iba rozhranie testovanej jednotky a nie jej vnútornú štruktúru.

3.1.2 Programovací jazyk SystemVerilog

SystemVerilog (IEEE 1800-2005) je programovací jazyk, ktorý sa radí medzi jazyky pre verifikáciu hardvéru (angl. *HVL - hardware verification language*). Na rozdiel od jazykov pre popis hardvéru (angl. *HDL - hardware description language*), ako napríklad Verilog alebo VHDL, jazyk SystemVerilog naviac obsahuje prostriedky pre:

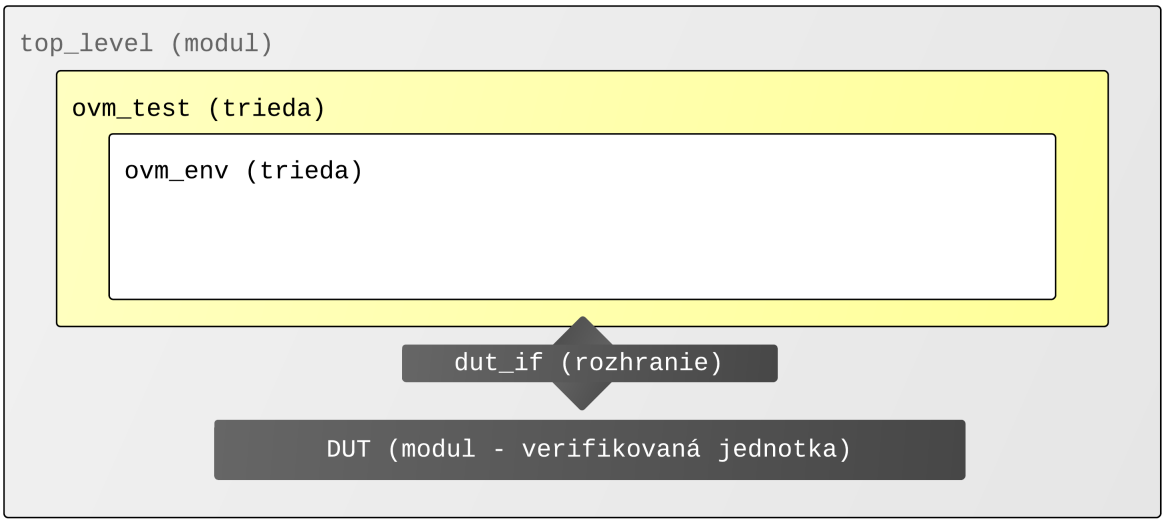
- náhodné generovanie stimulov,
- meranie funkčného pokrytia a pokrytia kódu,
- zložitejšie jazykové konštrukcie, najmä v súvislosti s objektovo orientovaným programovaním,
- multivláknovú a medziprocesovú komunikáciu,
- podporu pre HDL dátové typy, ako napríklad 4 stavové hodnoty v jazyku Verilog,
- kooperáciu s inými programovacími jazykmi (napríklad programovací jazyk C).

Kooperácia s inými programovacími jazykmi prebieha pomocou rozhrania DPI (angl. *Direct Programming Interface*). Toto rozhranie obsahuje dve izolované vrstvy. Jedna vrstva slúži pre jazyk SystemVerilog a druhá pre programovací, s ktorým sa bude spolupracovať. Rozhranie DPI môže byť použité napríklad pre zahrnutie referenčného modelu implementovaného v programovacom jazyku C do verifikačného prostredia.

Objektovo orientované programovanie (OOP) v spojení s jazykom SystemVerilog umožňuje vytváranie verifikačných prostredí na rôznej úrovni abstrakcie. Vďaka kľúčovým vlastnostiam OOP - polymorfizmu a dedičnosti je možné v jazyku SystemVerilog vytvárať znovupoužiteľné verifikačné prostredia.

3.2 Verifikačné prostredie

Verifikačné prostredie [2] sa skladá z verifikovanej jednotky DUT (angl. *Design Under Test*) a komponentov, ktoré okolo nej tvoria prostredie pre komunikáciu. Úlohy verifikačného prostredia sú: poskytovanie vstupov jednotke DUT, monitorovanie výstupov a ich porovnanie s referenčnými výsledkami.



Obrázok 3.1: Zloženie verifikačného prostredia

Znovupoužiteľné verifikačné prostredie pre rôzne testovacie scenáre je možné vytvoriť aj vďaka hierarchickému rozdeleniu prostredia na úrovne podľa abstrakcie. Tento prístup je možné vidieť na obrázku 3.1. Na najnižšej úrovni sa nachádza verifikovaná jednotka spolu so svojím rozhraním. Toto rozhranie zoskupuje signály pre komunikáciu s verifikovanou jednotkou a slúži pre prepojenie s komponentami verifikačného prostredia.

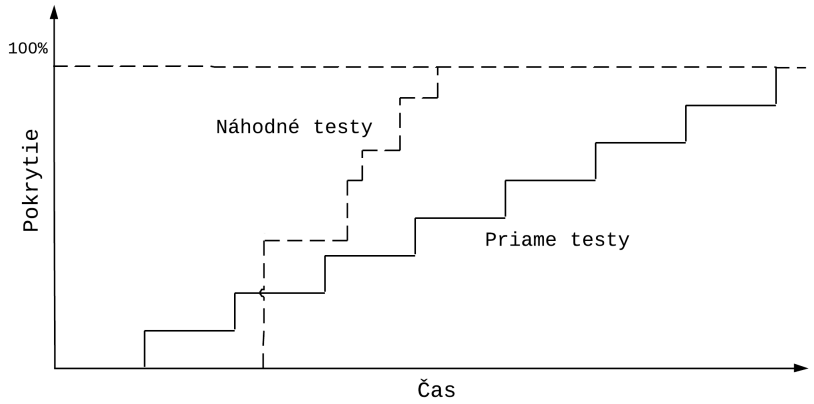
Na vyššej úrovni abstrakcie sa nachádza trieda `ovm_env` zoskupujúca verifikačné komponenty. V rámci tejto triedy je možné vytvoriť rôzne architektúry verifikačného prostredia na základe požiadaviek aktuálneho testovacieho scenára. Dynamická sada testov je zahrnutá na vyššej úrovni v triede `ovm_test`. Najvyššia úroveň abstrakcie zahŕňa modul `top_level`, kde sa vytvára inštancia jednotky DUT, jej prepojenie s verifikačným prostredím a dochádza k spusteniu testov.

Hierarchické členenie verifikačného prostredia vedie k intuitívnemu tvoreniu a rozširovaniu tohoto prostredia o nové komponenty na základe mnohých požiadaviek, ktoré sa pri verifikačnom procese vyskytnú.

Tvorenie verifikačného prostredia zjednodušuje použitie verifikačných metodík. Verifikačné metodiky sú popísané v poslednej podkapitole 3.3 v rámci tejto kapitoly.

3.2.1 Tvorba vstupných transakcií

Tvorba vstupných transakcií (angl. *Stimulus Generation Plan*) [10] zohráva dôležitú úlohu v rámci verifikačného prostredia. Pojem transakcia označuje prostriedok pre komunikáciu medzi komponentami a slúži pre prenos informácií. Na vyššej úrovni abstrakcie je viac transakcií zoskupených do jednej sekvencie. Tvorba vstupných transakcií zahŕňa generovanie alebo vytváranie vstupných vektorov pre jednotku DUT.



Obrázok 3.2: Rozdiel medzi priamymi a náhodnými testami

- **Priame testy** (angl. *Directed Testing*). Priame testy [10] sú základným spôsobom testovania počítačového systému. Pri tejto metóde obsahuje verifikačný plán konkrétne skupiny priamych testov, ktoré sú vždy zamerané na určitú časť funkčnosti testovaného systému. Proces overovania v jednom kroku zahŕňa vytvorenie vstupných vektorov, simuláciu jednotky DUT s týmito vektormi a následne manuálne vyhodnotenie výstupov jednotky a priebehu signálov. Výhodou tejto metódy sú okamžité výsledky, avšak pokrok pri pokrytí jednotky DUT je pomalý.
- **Testy pre generovanie náhodných stimulov s obmedzujúcimi podmienkami**. Pri zložitejších systémoch nie sú priame testy postačujúcim prostriedkom pre verifikáciu. Efektívnejšou technikou je generovanie náhodných transakcií [10] (angl. *Constrained-Random Stimulus Generation*). Generovanie nie je úplne náhodné, ale musí spĺňať určite požiadavky dané protokolom, alebo špecifikáciou. Pri takomto generovaní dochádza k odhaleniu nečakaných chýb a k rýchlejšiemu pokrytiu verifikovaného systému. Náhodné testovanie zníži počet potrebných testov a tým zvýši efektivitu verifikácie, čím sa urýchli celý verifikačný proces. Na záver testovania pomocou náhodného generovania transakcií je dobré pokryť doposiaľ neotestované časti systému pomocou priamych testov. Vplyv na pokrytie verifikovanej jednotky pri použití priamych testov a pri náhodne generovaných transakciách je možné vidieť na obrázku 3.2.

3.2.2 Kontrola výstupov

V rámci verifikačného prostredia identifikujeme ako ďalšiu dôležitú úlohu monitorovanie a kontrolu výstupov (angl. *Checker Plan*) [5] jednotky DUT. Kontrola spočíva v porovnaní pozorovaných výstupov s referenčnými výstupmi. Táto kontrola môže byť manuálna, automatizovaná a nezávislá od behu jednotlivých testov.

- **Scoreboarding.** *Scoreboarding* je technika, ktorá umožňuje dynamické predpovedanie výstupov jednotky DUT. Transakcie vstupujú do jednotky DUT a zároveň do transformačnej funkcie, ktorá aplikuje transformácie na vstupnú transakciu a vyprodukuje očakávané výstupy. Tieto výstupy sú uložené v dátovej štruktúre a neskôr sú použité pre porovnanie s výstupmi z jednotky DUT. Porovnanie v scoreboarde môže byť vykonané dvomi rôznymi spôsobmi:

  - **FIFO prístup.** FIFO (angl. *First-in-First-out*) prístup znamená, že výstup z DUT je porovnaný z predpovedaným výsledkom, ktorý je v tabuľke uložený ako prvý záznam. Tento prístup je možné používať tam, kde sú transkacie obsluhované v poradí a poradie výstupných transakcií nebude v rámci transformačnej funkcie zmenené.
  - **Porovnanie všetkých transakcií v tabuľke.** Po vyprodukovaní výstupnej transakcie z DUT je s touto transakciou porovnaná každá z predpovedaných výstupných transakcií uložených v tabuľke. Tento prístup je vhodný tam, kde sa transakcie môžu predbiehať a nezáleží na ich poradí pri obsluhe.

  Scoreboarding teda nepriamo zahŕňa ekvivalentnú funkčnosť k testovanému systému. Táto funkčnosť je naprogramovaná len na základe špecifikácie systému bez znalosti implementácie už vytvorenej implementácie systému. Funkčnosť overovaného systému v rámci verifikačného prostredia je obvykle uložená mimo scoreboardu, vo forme referenčného modelu (niekedy tiež býva označený ako golden model) alebo transformačnej funkcie.

- **Referenčný model** (angl. *Reference Model*). Referenčný model [8] zahŕňa funkčnosť ekvivalentnú k tej, ktorá je obsiahnutá v implementácii verifikovaného systému. Vstup, ktorý je aplikovaný na jednotku DUT je predaný takisto do referenčného modelu. Výstup z referenčného modelu je porovnaný s výstupom z jednotky DUT. Referenčný model je často tvorený zo špecifikácie architektúry systému a teda poskytuje rovnaké dátové transformácie a interakciu. Výhodou referenčného modelu je, že jednoduchou cestou poskytuje predpoveď výstupov. Dva základné typy referenčných modelov sú:

  - **Cycle-accurate** (angl. *Cycle-Accurate Reference Model*). Cycle-accurate znamená, že referenčný model a DUT produkujú výstupy v tých istých časových okamihoch, čo umožňuje rýchle a jednoduché porovnanie. Každý rozdiel vo výstupe môže byť nahlásený okamžite.
  - **Behaviorálne** (angl. *Behavioral Reference Model*). Behaviorálny referenčný model používa vyššiu mieru abstrakcie a určuje aké výstupy by mala jednotka DUT produkovať, no nešpecifikuje presné časové okamihy, kedy sa objavia na výstupe.

- **Offline kontrola** (angl. *Offline checking*). Offline kontrola sa používa pri predpovedi výstupov systému pred alebo po vykonaní simulácii systému. Pri predpovedi výstupov

systému pred simuláciou sa predpovedané výsledky uložia a pri behu simulácie sa dynamicky porovnávajú s pozorovanými výstupmi.

3.2.3 Kontrola pokrytia

Pokrytie [8] je metrika, ktorá nám v rámci verifikačného procesu poskytuje informácie o dosiahnutom pokroku. Táto metrika je zaznamenávaná počas simulácie a po jej skončení je vyhodnotená. Pre verifikátorov slúži pokrytie ako forma spätnej väzby. Rozlišujeme tri základné druhy pokrytia:

1. **Pokrytie kódu** (angl. *Code Coverage*) [9] definuje úseky kódu v zdrojových súboroch DUT, ktoré boli v priebehu simulácie vykonané. Vysoké pokrytie kódu síce nezaručuje dokončenie verifikácie, ale je žiadané kvôli čo najdôkladnejšiemu preskúmaniu kódu a odhaleniu možných chýb. Pokrytie kódu sa meria nasledujúcimi technikami:
   - pokrytie príkazov (angl. *Statement Coverage*),
   - pokrytie ciest (angl. *Path Coverage*),
   - pokrytie skokových príkazov (angl. *Branch Coverage*),
   - pokrytie podmienok (angl. *Condition Coverage*),
   - pokrytie výrazov (angl. *Expression Coverage*).
2. **Funkčné pokrytie**. Spätnú väzbu ohľadom pokrytia funkčných častí implementácie overovaného systému poskytuje funkčné pokrytie (angl. *Function Coverage*). Cieľom funkčného pokrytia je priniesť informáciu o dosiahnutom pokroku vo funkčnej verifikácie. Získaná spätná väzba informuje takisto o pokrytí legálnych stavov a okrajových podmienok systému. Niektoré techniky ako napríklad generovanie náhodných stimulov s obmedzujúcimi podmienkami dokážu zabezpečiť až 100% funkčné pokrytie, ktoré vedie k ukončeniu verifikačného procesu. Pre použitie funkčného pokrytia je potrebné vopred určiť, ktoré parametre a funkcie budú počas simulácie sledované.
3. **Pokrytie stavov konečného automatu** (angl. *Finite State Machine Coverage*). Jedná sa o merania navštívených stavov a využitých prechodov v konečnom automate počas simulácie.

V praxi sa najčastejšie používa funkčná verifikácia riadená pokrytím. Pri tomto prístupe sa používajú body pokrytia (angl. *Coverage Points*). Tieto body pokrytia zahŕňajú všetky časti DUT, ktoré chceme otestovať. Pri menej komplexných návrhoch môžeme otestovať všetky podporované operácie. Body pokrytia môžu byť signálové (angl. *Signal Coverpoints*), kedy nás zaujímajú transakcie v rámci jednotlivých signálov alebo dátové (angl. *Data Coverpoints*), kedy nás zaujímajú hodnoty a kombinácie hodnôt uložené v premenných DUT počas simulácie.

3.2.4 Kontrola pokrytia formálnych tvrdení

Dôležitou metódou, ktorá sa využíva v rámci verifikácie sú formálne tvrdenia (angl. *Assertions*). Sú to výroky týkajúce sa implementácie systému, ktoré musia byť vždy splnené. Príkladom ich použitia môže byť kontrola dodržiavania komunikačného protokolu na rozhraní verifikovanej jednotky.

Formálne tvrdenia sú výroky založené na predpokladoch [2] týkajúcich sa operačného prostredia jednotky DUT, verifikačného prostredia či na predpokladoch vychádzajúcich zo špecifikácie systému. Výrok sa môže skladať z logickej a z temporálnej časti. Logická časť je obvykle vyjadrená ako rovnica, zatiaľ čo temporálna časť špecifikuje časový okamih, kedy má rovnica platiť.

Formálne tvrdenia sa zapisujú pomocou špecializovaných jazykov (angl. *Assertion Language*). Príkladom je SVA (angl. *SystemVerilog Assertions*), ktorý ponúka prostriedky pre definíciu formálnych tvrdení. V jazyku SVA sú reprezentované úsekmi deklaratívneho kódu, ktoré kontrolujú vzťahy medzi signálmi v jednotke DUT, pričom kontrola môže byť jednorázová alebo periodická.

3.3 Verifikačné metodiky

Pojem metodika označuje súhrn praktík a návodov, metód, techník a nástrojov pre riešenie určitého problému. Verifikačné metodiky sú zamerané na tvorbu verifikačného prostredia pomocou znovupoužiteľných komponentov. Ich cieľom je vytvorenie verifikačného prostredia, ktoré bude efektívne, kvalitné a ľahko rozšíriteľné.

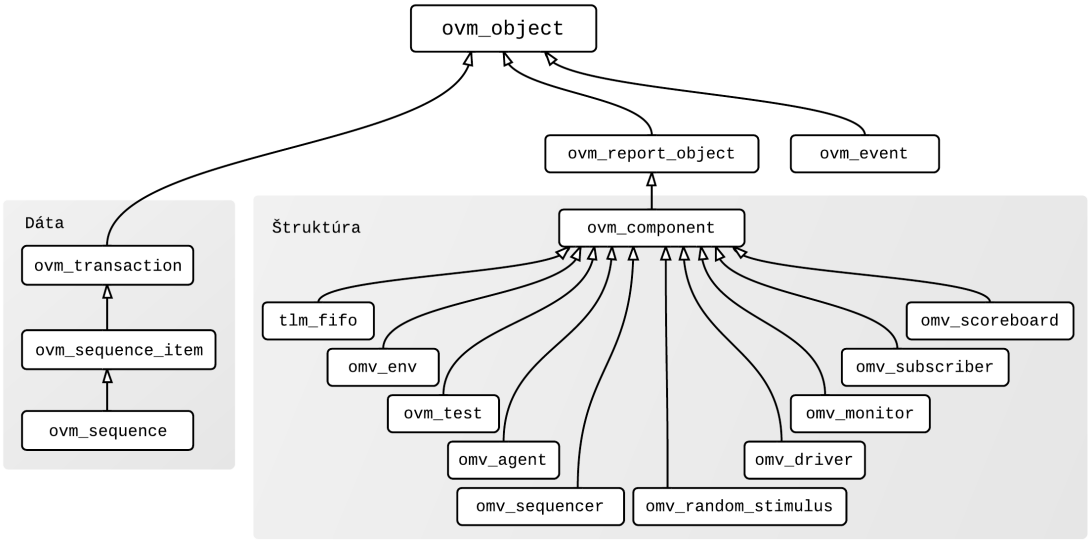
3.3.1 História verifikačných metodík

Jednou z prvých verifikačných metodík bola e metodika pre znovupoužiteľnosť označovaná ako eRM (angl. *ε Reuse Methodology*) uvedená v roku 2002 spoločnosťou Verisity Design. Táto metodika definovala postupy pre vytváranie konfigurovateľných a znovupoužiteľných komponentov a tiež konvencie pre písanie kódu, pomenovanie komponentov či architektúru verifikačného prostredia. Architektúra eRM metodiky slúžila spoločnosti Cadence ako podklad pri vytvorení metodiky URM (angl. *Universal Reuse Methodology*) pre jazyk SystemVerilog. Spoločnosť Mentor Graphics v roku 2006 vytvorila metodiku AVM (angl. *Advanced Verification Methodology*), ktorá prvá zahŕňala pokročilé techniky, ako napríklad generovanie náhodných stimulov s obmedzujúcimi podmienkami, funkčné pokrytie alebo formálne tvrdenia. Metodika bola implementovaná v SystemC (sada tried programovacieho jazyka C++ pre simuláciu riadenú udalosťami) a v jazyku SystemVerilog.

OVM (angl. *Open Verification Methodology*) je metodika, ktorá spája koncepty z vyššie uvedených dvoch metodík (URM a AVM). Vznikla v roku 2008 vďaka spolupráci spoločností Mentor Graphics a Cadence.

Metodika VMM (angl. *Verification Methodology Manual*) vznikla v roku 2005, ako výsledok spolupráce spoločností ARM a Synopsys. Pomocou tejto metodiky je možné vytváranie konfigurovateľných a znovupoužiteľných verifikačných komponentov s využitím pokročilých techník a princípov, ako napríklad verifikácia riadená funkčným pokrytím, formálna verifikácia alebo generovanie náhodných stimulov.

Štandardizovaná metodika UVM (angl. *Universal Verification Methodology*) je odvodená od metodiky OVM. V roku 2011 bola predstavená organizáciou Accelera. Metodika UVM je ako jediná podporovaná všetkými spoločnosťami skupiny EDA (angl. *Electronic Design Automation*). Metodika UVM oproti predchádzajúcej OVM prináša navyše vrstvu na úrovni registrov (angl. *Register Layer*), možnosť rozdelenia behu simulácie na viac fáz či mechanizmus sekvencií.



Obrázok 3.3: Hierarchia tried metodiky OVM

3.3.2 Metodika OVM

Metodika OVM je pravidelne aktualizovaná a voľne dostupná. Využíva princípy znovupoužiteľnosti a rozšíriteľnosti pri tvorbe verifikačného prostredia. Aktuálne vo verzii 2.1.2 (jún 2011) je dostupná spolu s metodikou UVM na stránkach Verification academy [17]. OVM ponúka mechanizmy pre spoluprácu VIP (angl. *Verification Intelligence Property*), modelov na transakčnej a RTL (angl. *Register Transfer Level*) úrovni a integráciu s inými často používanými jazykmi. Každý simulátor podporujúci štandard IEEE 1800 (angl. *Institute of Electrical and Electronics Engineers*) podporuje tiež OVM metodiku. Hierarchiu predpripravených tried, ktoré metodika OVM ponúka je možné vidieť na obrázku 3.3.

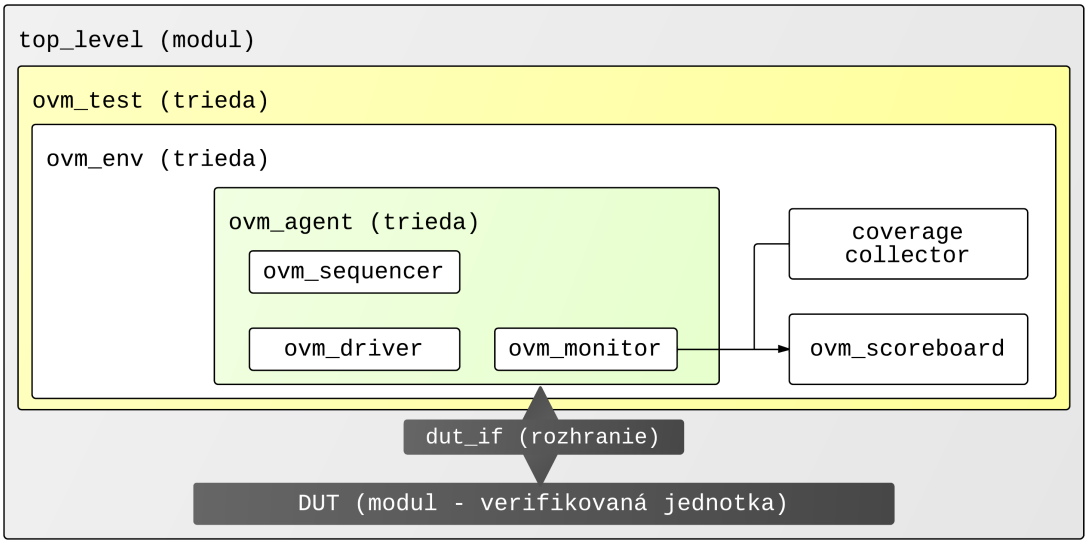
3.3.3 Komponenty verifikačného prostredia

Všeobecné hierarchické členenie verifikačného prostredia bolo popísané v podkapitole 3.2, preto sa táto časť venuje podrobnejšiemu popisu zloženia verifikačného prostredia v rámci metodiky OVM.

Komponenty verifikačného prostredia [7] by mali byť odvodené od triedy `ovm_component`, alebo `ovm_threaded_component`. Základné triedy pre tvorenie komponentov prostredia sú `ovm_test`, `ovm_env`, `ovm_monitor`, `ovm_driver`, `ovm_sequencer`, `ovm_virtual_sequencer`, `ovm_scoreboard`.

Zloženie verifikačného prostredia je možné vidieť na obrázku 3.4. Verifikačné prostredie môže obsahovať viacero komponentov odvodených od triedy `ovm_agent`, ktoré zabezpečujú komunikáciu s rozhraním jednotky DUT. Komponent agent zaobaľuje komponenty na nižšej úrovni abstrakcie, ako napríklad driver, monitor a sequencer. Pomocou prostriedkov dostupných v triede `ovm_agent` je možné ovládať rozhranie alebo aktivitu monitorov z komponentov na vyššej úrovni abstrakcie.

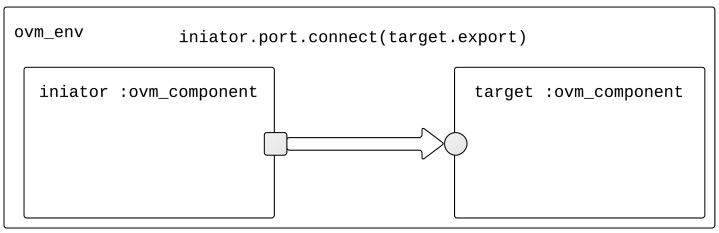
Driver je komponent odvodený od triedy `ovm_driver`, ktorá poskytuje funkčnosť potrebnú k interakcii s jednotkou DUT. Komponent monitor odvodený od triedy `ovm_monitor` zahŕňa funkčnosť potrebnú k pasívnemu pozorovaniu portu jednotky DUT. Monitor obsa-



Obrázok 3.4: Zloženie verifikačného prostredia podľa metodiky OVM

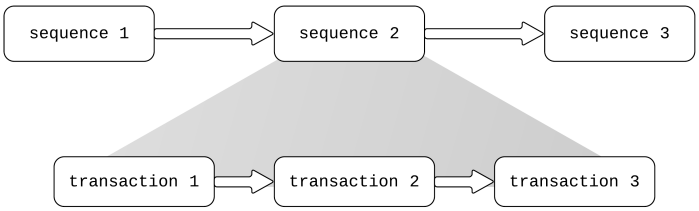
huje takisto prostriedky pre zber dát týkajúcich sa pokrytia. Ďalšou časťou komponentu agent je sequencer odvodený od triedy `ovm_sequencer`. Jeho úlohou je vytváranie stimulov, ktoré sú predávané do driveru. Sequencer môže byť riadený autonómne alebo pomocou komponentu sequencer nachádzajúceho sa v inom verifikačnom komponente. Zbieranie informácii týkajúcich sa pokrytia má na starosti komponent označený ako coverage_collector odvodený od triedy `ovm_subscriber`.

3.3.4 Prepojenie komponentov verifikačného prostredia



Obrázok 3.5: Prepojenie komponentov pomocou portov.

Komponenty verifikačného prostredia sú spojené pomocou prepojení označených ako porty a exporty. Tie prepojenia sú štandardné TLM (angl. *Transaction Level Modeling*) rozhrania. Informácie, ktoré umožňujú sledovať pokrytie sú získavané zo špeciálnych portov určených pre analýzu transakcií (`analysis_port`). Prepojenie komponentov pomocou portov ilustruje obrázok 3.5.

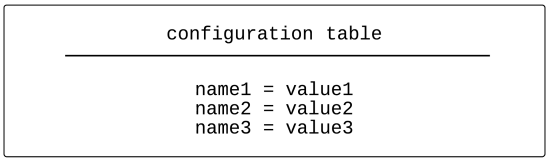


Obrázok 3.6: Forma komunikácie medzi komponentami.

3.3.5 Komunikácia medzi komponentami

Pre komunikáciu sú použité sekvencie. Sekvencie môžu byť zanorené, vrstvené alebo virtuálne a môžu obsahovať transakcie. Tento princíp je možné vidieť na obrázku 3.6. Obsah transakcií môže byť generovaný náhodne, ale zároveň musí spĺňať obmedzujúce podmienky, ktoré sú dané komunikačným protokolom na rozhraní. V metodike OVM sú pre komunikáciu dostupné triedy `ovm_sequence`, `ovm_sequence_item` a `ovm_transaction`.

3.3.6 Konfigurácia



Obrázok 3.7: Konfiguračná tabuľka.

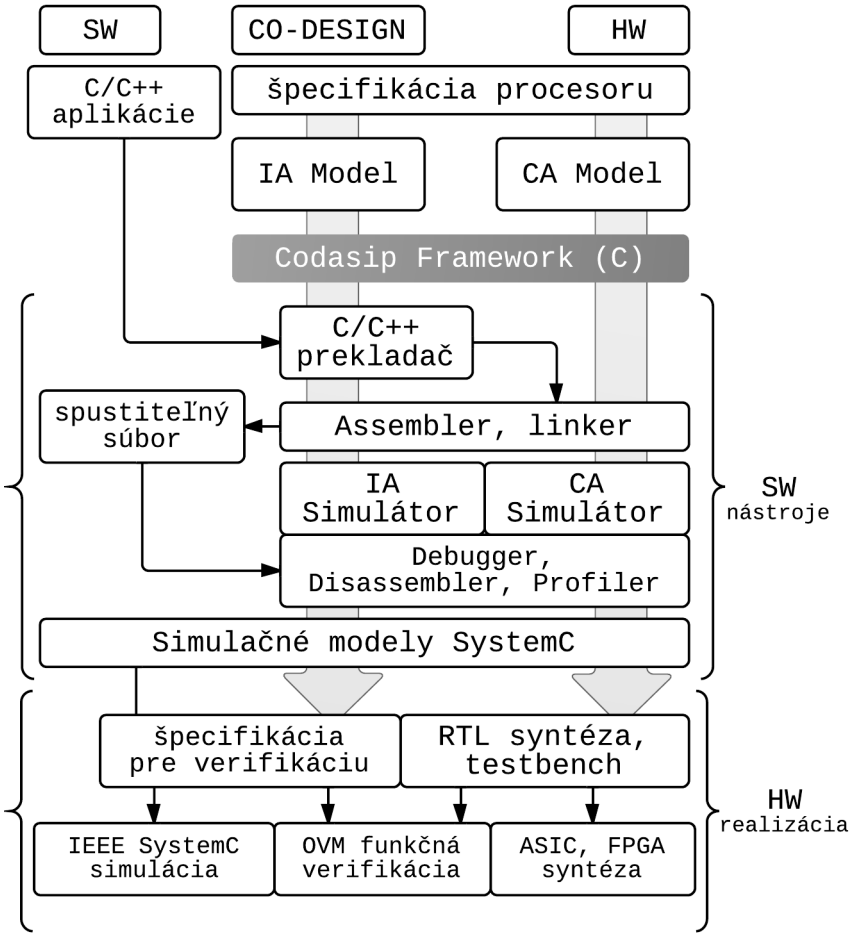
Znovupoužiteľnosť komponentov verifikačného prostredia je zabezpečená aj pomocou mechanizmu konfigurácie. Ten zahŕňa konfiguračnú tabuľku, kde sú uložené konfiguračné informácie pre odvodené komponenty na nižšej úrovni abstrakcie. Obrázok 3.7 dokumentuje tento mechanizmus.

Vďaka tomuto mechanizmu je možné prispôsobiť a nastaviť komponent verifikačného prostredia podľa prostredia v akom sa momentálne nachádza.

Kapitola 4

Systém Codasip

V tejto kapitole je popísané zloženie systému Codasip, pomocou ktorého bol vygenerovaný procesor Codix, ktorý je verifikovaný v tejto práci. Jednotlivé podkapitoly sa venujú hlavným modulom systému Codasip, pričom záver kapitoly obsahuje stručný popis procesora Codix a jeho rozhrania.



Obrázok 4.1: Schéma systému Codasip

Systém Codasip [3] zahŕňa integrované vývojové prostredie, ktoré ponúka návrhárom možnosť vytvoriť aplikačne špecifické procesory (angl. *ASIP - Application-Specific Instruction-Set Processors*) a multiprocesorové systémy na čipe (angl. *MPSoC Multiprocessor Systems on Chip*). Aplikačne špecificky inštrukčné procesory sú základným stavebným blokom vstavaných systémov, ktoré ovládajú moderné elektronické systémy.

Výhodou systému Codasip je rýchlosť návrhu takýchto procesorov a automatizácia úloh, ktoré by inak museli byť vykonané manuálne. Návrh začína vytvorením popisu architektúry procesora na vysokej úrovni abstrakcie v jazyku CodAL. Popis jazyka CodAL nasleduje v ďalšej podkapitole. Pomocou systému Codasip sú z popisu v jazyku CodAL automaticky generované simulačné nástroje, syntentizovateľný RTL popis a prostredie pre funkčnú verifikáciu. Obrázok 4.1 zobrazuje schému systému Codasip.

4.1 Jazyk CodAL

Jazyk CodAL [4] bol navrhnutý pre rýchle prototypovanie procesorov typu ASIP a MPSoC. Tento jazyk je možné zaradiť medzi jazyky pre popis architektúr. Jazyk CodAL podporuje súbežný vývoj softvéru a hardvéru (angl. *Hardware/Software Codesign*). Ďalej umožňuje automatické vytvorenie implementácie mikroarchitektúry v hardvéri z generovaného popisu v jazyku VHDL.

Kľúčovou vlastnosťou jazyka CodAL je možnosť použiť referenčný model navrhnutého procesora vo forme IA (angl. *Instruction Accurate*) modelu a zaručenie jeho ekvivalencie s komplexným CA (angl. *Cycle Accurate*) modelom.

Definícia procesora napísaná v jazyku CodAL prechádza kontrolou a následne je preložená do internej reprezentácie vo formáte XML. Načítaný XML popis je použitý pre generovanie prekladača jazyka C, assembleru, disassembleru, spojovacieho programu, viacerých typov simulátorov a tiež pre generovanie RTL reprezentácie. Tento postup je veľmi rýchly a umožňuje rýchlu kontrolu návrhu (angl. *DSE - Design Space Exploration*), kedy návrhár môže zvážiť rôzne možnosti a kompromisy pri návrhu architektúry procesora.

Základné informácie, ktoré jazyk CodAL popisuje sú: inštrukčná sada procesora, model časovania, model správania a štrukturálny model.

4.2 Nástroje pre preklad

Z popisu architektúry procesora sú generované nástroje pre preklad jazyka C a C++. Prenositeľný C/C++ prekladač je založený na rozšírenom LLVM (angl. *Low Level Virtual Machine*) frameworku. Spolu s prekladačom je k dispozícii rozsiahla sada testov.

Tento prenositeľný prekladač prekladá aplikáciu z programovacieho jazyka C/C++ do programu vo formáte GNU assembler. Takto reprezentovaný program je pomocou nástroja assembler preložený do objektového súboru. Nakoniec nástroj linker spojí viacero objektových súborov a vytvorí z nich jeden spustiteľný súbor.

4.3 Simulácia a ladenie

Dôležitým nástrojom generovaným pomocou systému Codasip je simulátor. Návrhár má možnosť vybrať si z rôznych variantov generovaných simulátorov odlišujúcich sa časom potrebným na vytvorenie a rýchlosťou simulátora. V simulátore je možnosť sledovať užitočné

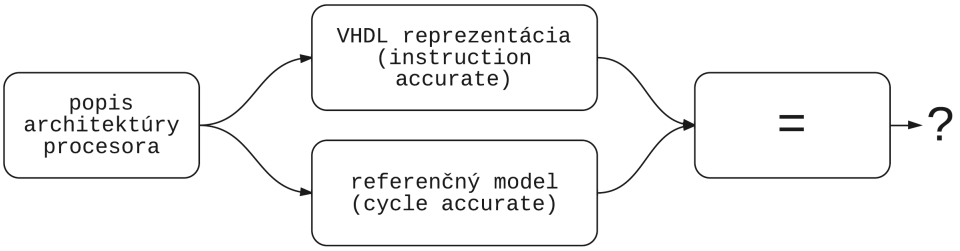
informácie, ako napríklad hodnoty registrov, ktoré prispievajú k skoršiemu odhaleniu chýb v návrhu.

Pokiaľ sú potrebné podrobnejšie informácie o procese simulácie, tak je možné využiť nástroj profiler. Ten sleduje a zaznamenáva dôležité informácie a štastistiky. Ich príkladom je najviac alebo najmenej používaná inštrukcia, práca s rýchlou vyrovnávacou pamäťou, štatistika volania funkcií, využitie zdrojov a iné.

4.4 Generovanie RTL reprezentácie

Ak je architektúra procesora stabilná je možné vygenerovať syntetizovateľnú RTL reprezentáciu procesora. Spolu s vygenerovaním sady testov je tiež možné generovať podporu pre JTAG (angl. *Join Test Action Group*) ladiace rozhranie. Správanie procesora v generovaných simulátoroch by malo odpovedať správaniu procesora v RTL reprezentácii.

4.5 Funkčná verifikácia



Obrázok 4.2: Schéma funkčnej verifikácie pre procesory vytvorené systémom Codasip.

Cieľom nástrojov vygenerovaných pre funkčnú verifikáciu je overiť ekvivalenciu medzi IA modelom, z ktorého je generovaný prekladač jazyka C/C++ a CA modelom, na základe ktorého sa generuje RTL reprezentácia procesora. Tento princíp približuje obrázok 4.2. Prostredie funkčnej verifikácie je navrhnuté v súlade s metodikou OVM. Pre svoje efektívne použitie pri návrhu procesora obsahuje rozsiahlu sadu testov, ktorými je možné odhaliť nezhody medzi vyššie spomenutým IA a CA modelom. Spätnou väzbou funkčnej verifikácie je pokrytie kódu, ktoré je v generovanom prostredí taktiež k dispozícii.

4.6 Procesor Codix

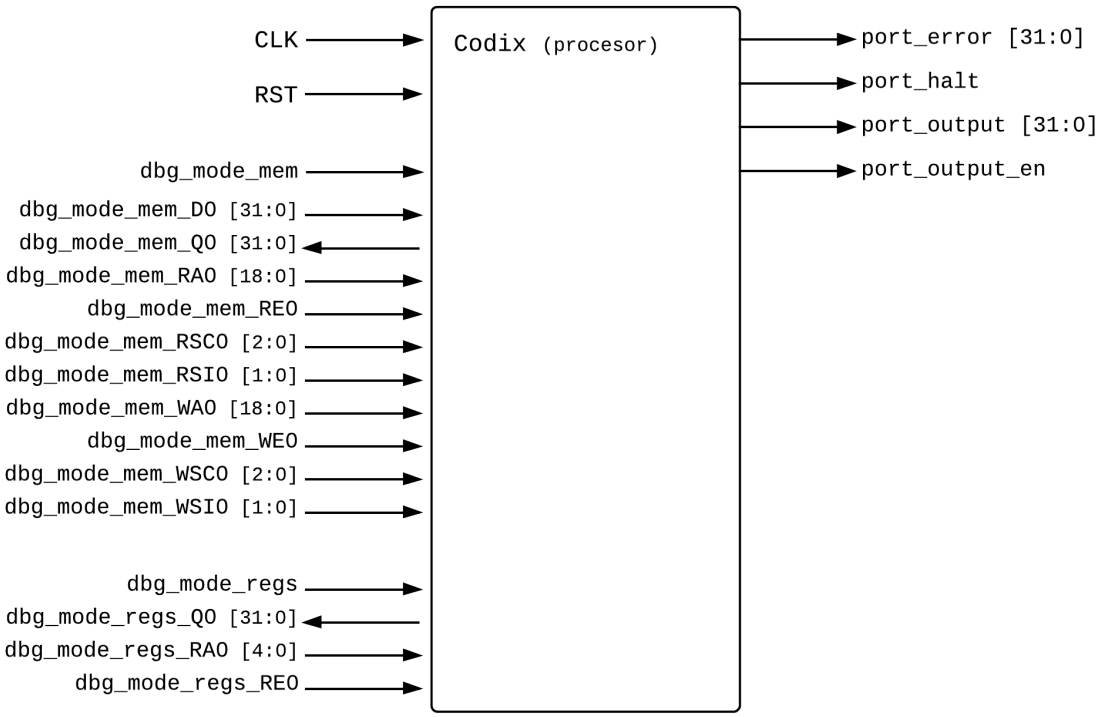
4.6.1 Špecifikácia procesora

Verifikovaný procesor má názov Codix. Bol navrhnutý pomocou systému Codasip. Jedná sa o procesor s veľkosťou slova 32 bitov. Využíva sa v kombinácii s integrovaným obvodom FPGA alebo ako aplikačne špecifický procesor.

Procesor obsahuje jednu pamäť, ktorá slúži pre uloženie programu aj dát. Pamäť má dva dátové porty, jeden pre čítanie, druhý pre zápis a jej veľkosť je voliteľná vzhľadom na použitie procesora. Registrové pole, ktoré procesor obsahuje sa skladá z 32 prvkov, pričom každý prvok má veľkosť 32 bitov.

Procesor podporuje rozšírenie pomocou zberníc podľa účelu jeho použitia a obsahuje tiež sedemstupňovú linku pre zreťazenie spracovania inštrukcií.

Inštrukčná sada procesora obsahuje štandardné aritmeticko-logické operácie, inštrukcie pre prácu s pamäťou, prerušeniami, skokové a špeciálne inštrukcie.



Obrázok 4.3: Rozhranie procesora Codix

4.6.2 Rozhranie procesora

Rozhranie procesora Codix je možné vidieť na obrázku 4.3. Pre synchronizáciu činnosti procesora slúžia signály `CLK` a `RST`. Práca s pamäťou procesora je možná pomocou signálov s prefixom `dbg_mode_mem*`. Pri zápise alebo čítaní obsahu pamäte je potrebné nastaviť adresu signál `RAO` (`WAO`) a povoľovací signál `REO` (`WEO`). Dáta sa pri čítaní následne objavia na výstupnom signály `QO`. Pre zápis dát slúži signál `DO`. Signál `WSCO` (`RSCO`) označuje počet podblokov, ktoré budú do pamäti zapisované alebo z pamäti čítané a signál `WSIO` (`RSIO`) označuje index zapisovaného alebo čítaného podbloku.

Čítanie obsahu registrového poľa je možné pomocou skupiny signálov vyvedenej na rozhranie procesora s prefixom `dbg_mode_regs*`. Pri čítaní je potrebné nastaviť povoľovací signál `REO` a adresu `RAO`. Následne sa dáta objavia na výstupnom signály `QO`.

Výstup procesora je zabezpečený pomocou signálu `port_output`, ktorého platnosť indikuje signál `port_output_en`. Informácie v prípade chybového stavu je možné vyčítať zo signálu `port_error`.

Pokiaľ procesor narazí pri vykonávaní programu na špeciálnu ukončujúcu inštrukciu halt dôjde k nastaveniu signálu `port_halt`.

Kapitola 5

Akcelerácia funkčnej verifikácie

Verifikačné prístupy založené na simulácii, medzi ktoré patrí aj funkčná verifikácia ponúkajú možnosť sledovať vnútorné správanie a signály testovaného systému. Problémom simulácie je však jej nízka rýchlosť. Pre implicitne paralelné hardvérové systémy je softvérová simulácia extrémne pomalá v porovnaní s rýchlosťou reálneho hardvéru [18].

Rozdiel medzi rýchlosťou softvérovej simulácie a reálneho hardvéru je výzvou pre vedecké tímy ale aj pre komerčné spoločnosti, pretože pomalý priebeh funkčnej verifikácie môže mať negatívny vplyv na čas uvedenia výrobku na trh (angl. *Time-to Market*).

V tejto kapitole budú popísané existujúce riešenia pre hardvérovú akceleráciu a výber riešenia použitého v tejto práci.

5.1 Existujúce riešenia

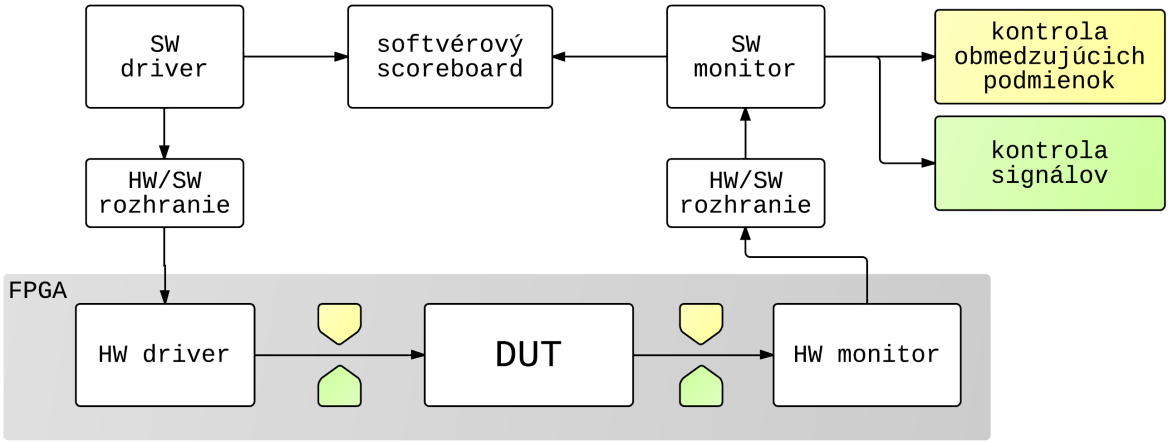
Existujú techniky a riešenia, ktoré sa zaoberajú akceleráciou funkčnej verifikácie. Príkladom je technológia od Mentor Graphics nazvaná `Veloce2` [16] umožňujúca akceleráciu funkčnej verifikácie pomocou syntézy jednotky DUT a jej následnej emulácie. Výhodou tohoto prístupu je zrýchlenie procesu verifikácie, zatiaľ čo zostáva možnosť sledovať signály. Maximálna frekvencia tohoto emulátora je $1,5$ MHz, čo nemusí byť postačujúce pre niektoré aplikácie využívajúce vysoko rýchlostné rozhrania.

Ďalším existujúcim riešením je `SEmulator` [11], ktorý umožňuje akceleráciu funkčnej verifikácie pomocou integrovaného obvodu `FPGA` (angl. *Field Programmable Gate Array*). Nevýhodou je strata možnosti sledovania signálov jednotky DUT.

Zaujímavým komerčným riešením je produkt `TBA` (angl. *Transaction Based Acceleration*) od spoločnosti Cadence [15], ktorý je súčasťou balíka `Cadence Incisive` pre akceleráciu a emuláciu. Toto riešenie ponúka 100 až 10000 násobné urýchlenie v rámci RTL simulácie pri počte $2x10^6$ hradiel. K dispozícií je tiež `DirectC` rozhranie, ktoré umožňuje spoluprácu s jazykmi C a C++. Hlavnými výhodami tohoto riešenia sú: dostupnosť signálov a transakcií, rýchlosť emulácie (1MHz+), flexibilné prostredie umožňujúce ladenie a jednoduchú tvorbu verifikačného prostredia a tiež profiler pre výkon, ktorý poskytuje informácie o slabých miestach vhodných pre akceleráciu.

Žiadne z týchto riešení však nie je voľne dostupné a šíriteľné a väčšinou sa jedná o komerčné produkty. Preto sa ako výhodná možnosť ponúka použitie voľne dostupnej platformy pre akceleráciu funkčnej verifikácie pomocou integrovaného obvodu `FPGA` s názvom `HAVEN` [6].

5.2 Akceleračná platforma HAVEN



Obrázok 5.1: Platforma pre akceleráciu funkčnej verifikácie - HAVEN

Platforma `HAVEN` (angl. *Hardware Accelerated Verfication ENvironment*) sa zamierava na nevýhodu funkčnej verifikácie, ktorou je nízka rýchlosť simulácie komplexných počítačových systémov. Pomocou presunu jednotky DUT a okolitých komponentov do prostredia hardvéru, konkrétne na integrovaný obvod FPGA je možné dosiahnuť výrazné zrýchlenie. Architektúru verifikačného prostredia platformy `HAVEN` ilustruje obrázok 5.1.

Pri práci s platformou `HAVEN` je možné využiť akcelerovanú alebo neakcelerovanú verziu. Pri neakcelerovanej verzii prebieha verifikácia v softvérovom simulátore, zatiaľ čo pri akcelerovanej verzii sa využíva hardvérová akcelerácia. Tieto odlišné verzie majú uplatnenie v rôznych fázach verifikačného procesu.

Akcelerovaná verzia má význam pri komplexných systémoch a značne urýchľuje pomalý proces simulácie. Počet transakcií v komplexných systémoch sa pohybuje rádovo v miliónoch. Časti popisujúce správanie navrhnutého systému, ako napríklad plánovanie testovacích vektorov, generovanie náhodných stimulov alebo scoreboarding ostávajú v softvérovom simulátore. Takéto rozdelenie jednotlivých komponentov verifikačného prostredia je možné vďaka návrhu založenom na použití transakcií, ktorý je súčasťou verifikačných metodík ako OVM alebo UVM. Signály sú napriek umiestneniu komponentov v špecializovanom hardvéri prístupné pre verifikátorov. Pomocou tejto platformy je možné dosiahnuť až 100000-násobné zrýchlenie funkčnej verifikácie.

Toto verifikačné prostredie ponúka knižnicu predpripravených základných a rozšírených verifikačných komponentov, ktoré sú organizované do balíčkov.

Kapitola 6

Návrh verifikačného prostredia pre procesory

Verifikačné prostredie, ktorého zloženie bolo popísané v podkapitole 3.2 tvorí základ funkčnej verifikácie. Pri jeho analýze a návrhu je potrebné vychádzať z požiadaviek kladených na priebeh, výsledky a rýchlosť verifikácie.

Priebeh verifikácie zahŕňa poskytovanie vstupov pre verifikovanú jednotku, sledovanie činnosti verifikovanej jednotky, monitorovanie jej výstupov a na záver porovnanie výstupov verifikovanej jednotky a referenčného modelu.

V prípade verifikácie procesora Codix je vstupom program v binárnej reprezentácii skladajúci sa z inštrukcií, ktorý je nahraný do pamäte procesora. Procesor potom zaháji svoju činnosť a počas spracovávania inštrukcií mení priebežne obsah pamäte a registrového poľa. Obsah pamäte a registrovej zložky je vyčítaný jeden krát, a to po ukončení činnosti procesora a nie priebežne pri spracovávaní inštrukcií. Tento prístup vedie k menšej réžii na výstupnom rozhraní procesora, a zároveň k väčšej efektivite komunikácie. Výstupom činnosti procesora je teda unikátny obsah pamäte a registrového poľa.

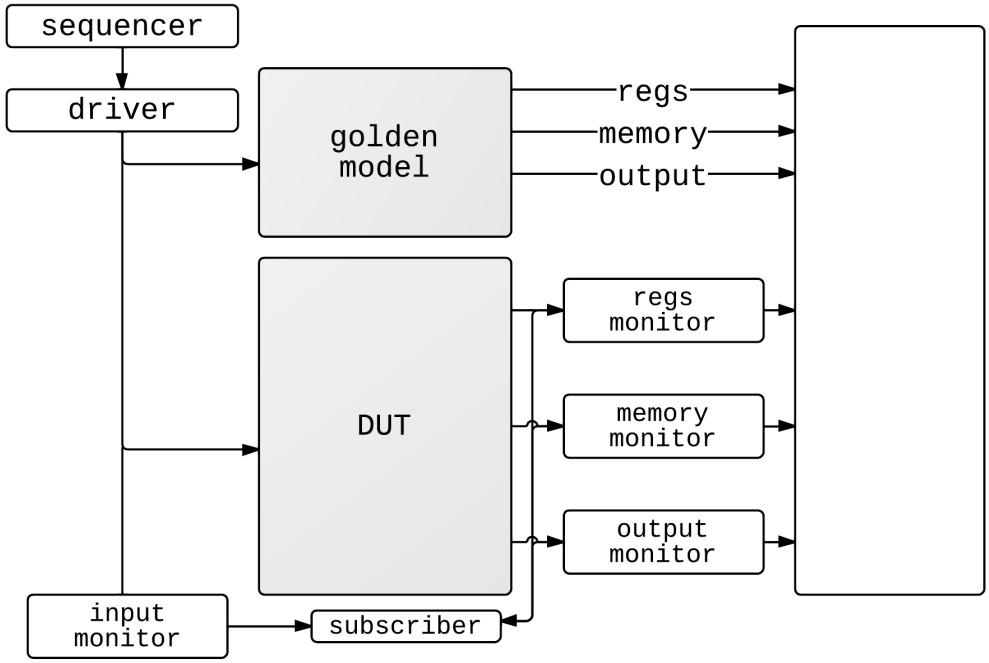
Výsledkom verifikácie je odpoveď na otázku, či sa zhodujú výstupy verifikovanej jednotky s referenčnými výsledkami. V prípade procesora sa jedná o jeho výstupy a výstupy dostupného referenčného modelu.

Najdôležitejšou z požiadaviek pri tomto projekte bolo skrátenie času potrebného pre verifikáciu procesora. Ako už bolo spomenuté v kapitole 5 - softvérová simulácia hadvéru je výrazne pomalšia ako jej alternatíva s použitím hardvéru pre akceleráciu. Práve preto sa ako vhodné riešenie ukazovalo vytvorenie akcelerovanej verzie verifikačného prostredia. Táto verzia urýchli priebeh verifikácie pomocou presunu procesora a potrebných komponentov verifikačného prostredia do hardvéru.

Akceleračná platforma `HAVEN`, ktorú využíva táto práca pre akceleráciu funkčnej verifikácie, umožňuje výber medzi pomalšou softvérovou verziou verifikačného prostredia a akcelerovanou verziou využívajúcou hardvér, pričom priebeh verifikácie je totožný. Tento prístup zachováva možnosť reprodukovania situácie, kedy dôjde pri verifikácii k odhaleniu chyby. Takú situáciu je možné potom pohodlne preskúmať v softvérovej verzii prostredia simulátora.

Princíp rozdelenia verifikačného prostredia na akcelerovanú a neakcelerovanú verziu bol zahrnutý do návrhu tejto práce na základe vyššie vymenovaných výhod tohoto riešenia a požiadaviek na verifikačný proces. Výsledné verifikačné prostredie by malo zahŕňať obidve verzie a jednoduchú možnosť výberu medzi nimi.

6.1 Neakcelerovaná verzia



Obrázok 6.1: Automaticky generované verifikačné prostredie pre procesor Codix.

Pre verifikáciu procesora bolo k dispozícii automaticky generované softvérové verifikačné prostredie pomocou systému Codasip. Toto prostredie bolo navrhnuté s využitím metodiky OVM. Využíva sadu testovacích vstupných programov, ktoré sú postupne v rámci jednotlivých testovacích scénarov posielané do pamäte procesora aj referenčného modelu. Výstupy z referenčného modelu a procesora sú privedené do scoreboardu, ktorý ich porovnáva a oznamuje prípadné nezhody. Funkčné pokrytie v rámci tohoto prostredia je vyhodnocované na základe pokrytia inštrukčnej sady vo vstupných testovacích programoch. Zloženie generovaného prostredia je možné vidieť na obrázku 6.1.

6.2 Akcelerovaná verzia

Návrh akcelerovanej verzie verifikačného prostredia vychádza zo všeobecného návrhu verifikačného prostredia v platforme `HAVEN`. Ten sa delí na dve základné časti: softvérovú a hardvérovú. Softvérová časť obsahuje verifikačné komponenty, ktorých funkciou je tvorba vstupov pre verifikovanú jednotku, mechanizmy pre kontrolu výstupov a sledovanie pokrytia. Referenčný model sa tiež nachádza v softvérovej časti. V hardvérovej časti sa nachádza verifikovaná jednotka a komponenty potrebné pre komunikáciu s jej rozhraním. Vďaka presunu verifikovanej jednotky do hardvérovej časti na integrovaný obvod FPGA je možné dosiahnuť výrazné urýchlenie priebehu testov.

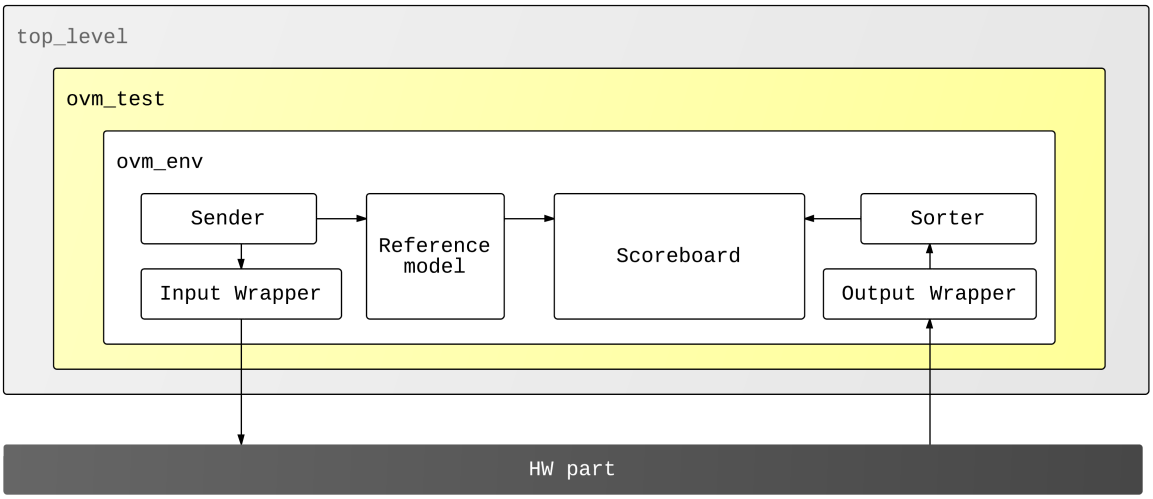
6.2.1 Komunikácia medzi softvérom a hardvérom

Medzi softvérovou a hardvérovou časťou prebieha komunikácia pomocou transakcií. Na vstupnej strane identifikácia nie je potrebná, pretože komunikácia prebieha medzi dvomi bodmi. Preto sa do komponentu `program_driver` v hardvérovej časti posielajú transakcie, ktoré obsahujú iba dátovú časť.

Výstup z hardvérovej časti musí obsahovať identifikáciu odosielateľa, ktorá je odosielaná pred dátovými transakciami v podobe jednej kontrolnej transakcie označujúcej zdroj dát.

6.2.2 Softvérová časť

Verifikačné prostredie v softvérovej časti je vytvorené v jazyku SystemVerilog. Jeho základom je automaticky generované prostredie využívajúce metodiku OVM, ktorej princípy sú popísané v podkapitole 3.3.2. Toto prostredie je rozšírené o komponenty z platformy `HAVEN`, ktoré slúžia pre komunikáciu s hardvérovou časťou. V rámci softvérovej časti je možné identifikovať nasledujúce funkcie: načítanie vstupného programu pre procesor uloženého v binárnej forme, odoslanie programu do hardvérovej časti po ukončení spracovania programu v procesore, prijatie obsahu pamäte, registrového poľa a výstupu z hardvérovej časti a porovnanie výstupov referenčného modelu s výstupmi procesora.



Obrázok 6.2: Softvérová časť akcelerovanej verzie verifikačného prostredia.

Návrh verifikačného prostredia v softvérovej časti je možné vidieť na obrázku 6.2.

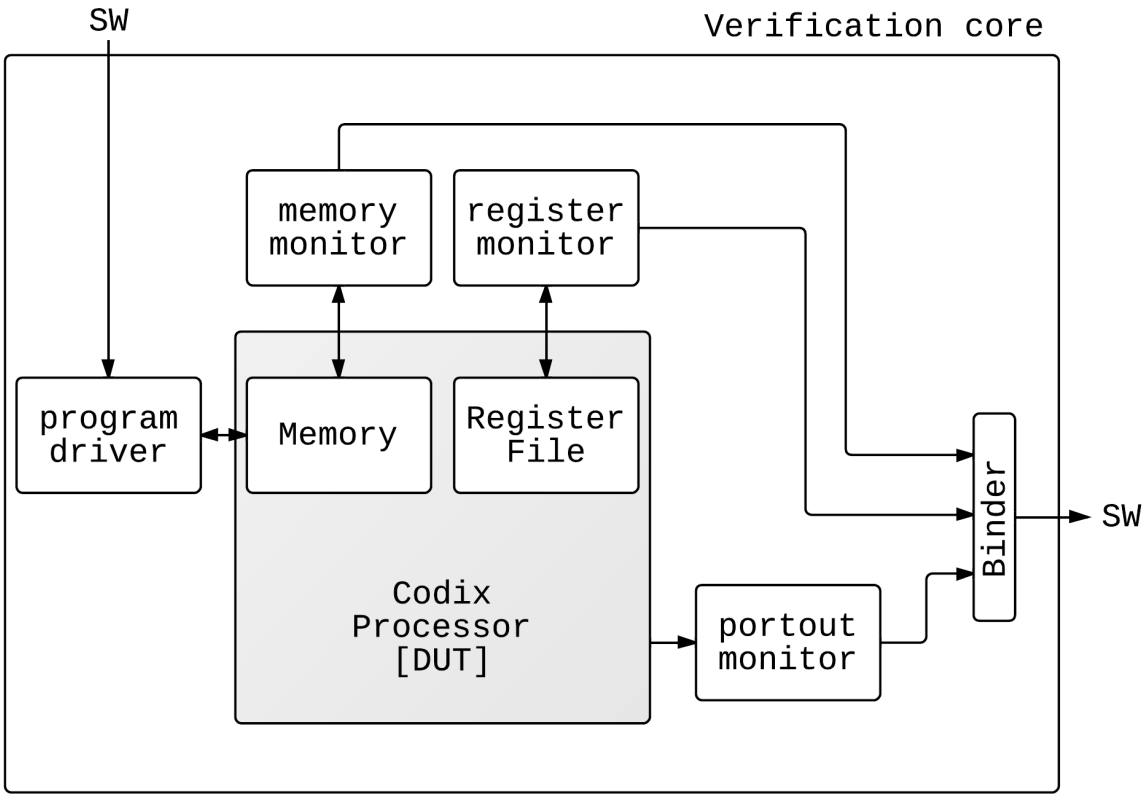
- `Sender` - načítava vstupný program v binárnej reprezentácii, kde jeden riadok programu reprezentuje jednu inštrukciu. Riadky vstupného programu sú odoslané do komponentu `Input Wrapper`.
- `Input Wrapper` - otvára `DMA` (angl. *Direct Memory Access*) kanál pre komunikáciu z hardvérom. Ďalej posiela transakcie obsahujúce vstupný program do hardvéru cez DPI rozhranie.
- `Referenčný model` - obsahuje funkčnosť ekvivalentnú s verifikovanou jednotkou implementovanú v programovacom jazyku C. K verifikačnému prostrediu je pripojený pomocou rozhrania DPI.

- `Scoreboard` - porovnáva výstupy z referenčného modelu s výstupmi, ktoré prídu z hardvérovej časti z verifikovanej jednotky.
- `Output Wrapper` - zatvára DMA kanál po prijatí všetkých transakcií z hardvéru. Prijíma `NetCOPE` transakcie z hardvéru a preposiela ich do komponentu sorter.
- `Sorter` - prijaté transakcie z hardvéru rozdeľuje a posiela do scoreboardu podľa hlavičky, kde je identifikácia odosielateľa.

6.2.3 Hardvérová časť

Komponenty tvoriace hardvérovú časť sú naprogramované v jazyku `VHDL`. Komponenty dodržiavajú jednotné rozhranie pre komunikáciu, ktoré umožňuje univerzálny a znovu použiteľný návrh verifikačného prostredia.

Základom hardvérovej časti je komunikácia s verifikovanou jednotkou. Pre tento účel je tu prítomný komponent, ktorý poskytuje vstup verifikovanej jednotke a viacero komponentov, ktoré monitorujú výstupy verifikovanej jednotky. Signály na rozhraniach komponentov moniturujúcich výstup verifikovanej jednotky musia byť spojené do jedného rozhrania, cez ktoré budú odoslané do softvérovej časti. Túto úlohu vykonáva komponent `Binder`.



Obrázok 6.3: Hardvérová časť akcelerovanej verzie verifikačného prostredia.

Návrh verifikačného prostredia nachádzajúceho sa hardvérovej časti je na obrázku 6.3.

- `Program Driver` - má na starosti komunikáciu s procesorom pred zahájením jeho činnosti. Od začiatku svojej činnosti udržuje procesor v aktívnom resete, vďaka čomu

procesor nezaháji svoju činnosť skôr ako bude vstupný program nahratý do jeho pamäte. Pomocou rozhrania procesora nahráva vstupný program do programovej časti jeho pamäte. Po nahratí programu do pamäte procesora je reset deaktivovaný a procesor zaháji svoju činnosť. Opätovné spustenie komponentu program driver je možné až po detekcii signálu indikujúceho ukončenie činnosti procesora a čítania jeho výstupov od komponentu `Memory Monitor`.

- `DUT` - procesor Codix - verifikovaná jednotka dostupná vo forme zdrojových súborov v programovacom jazyku VHDL.
- `Halt Monitor` - po detekcii inštrukcie halt distribuuje signál halt do ostatných monitorov pri výstupných rozhraniach procesoru. Okrem toho nastaví reset procesora na aktívnu hodnotu, čím sa zaručí nemenný obsah jeho pamäte a registrovej zložky. V tom momente je možné zahájiť činnosť monitorov.
- `Portout Monitor` - pri aktívnej hodnote signálu `port_output_en` posiela na výstup hodnotu signálu `port_output`. Pred dáta vždy pripojí hlavičku s identifikáciou odosielateľa.
- `Register Monitor` - po detekcii inštrukcie halt spustí svoju činnosť a vyčíta obsah registrovej zložky pomocou signálov vyvedených na rozhranie procesora. Pred dátami pripája svoju identifikáciu.
- `Memory Monitor` - aktivuje sa po detekovaní ukončenia činnosti komponentu `Register Monitor`. Vyčíta obsah pamäte procesora a dáta spolu s identifikáciou posiela na výstup. Ukončenie svojej činnosti potom oznámi komponentu `Program Driver`, ktorý môže zahájiť nahrávanie ďalšieho vstupného programu do pamäte procesora.
- `Binder` - spája výstupné rozhrania z monitorov do jedného rozhrania, ktoré komunikuje so softvérovou časťou verifikačného prostredia.

Kapitola 7

Implementácia

V tejto kapitole sú zhrnuté špecifické technológie použité pri implementácií verifikačného prostredia akcelerovanej verzie.

7.1 Softvérová časť

7.1.1 OVM

Softvérová časť akcelerovanej verzie je implementovaná v jazyku SystemVerilog s použitím metodiky OVM. Jej použitie vedie k jednoduchej a efektívnej architektúre verifikačného prostredia. Pomocou parametra pre test je možné vybrať akcelerovanú alebo neakcelerovanú verziu verifikačného prostredia.

Na najvyššej úrovni sa nachádza modul `top_level`. Na nižšej úrovni je to trieda pre test odvodená od triedy `ovm_test` a modul pre verifikovanú jednotku. Modul pre verifikovanú jednotku nie je v tomto prípade potrebný, pretože tá sa nachádza v hardvérovej časti verifikačného prostredia.

Test zahŕňa triedu pre verifikačné prostredie odvodenú od triedy `ovm_env`. V nej dochádza k vytvoreniu a prepojeniu komponentov. Toto prepojenie je implementované pomocou fronty odvodenej od triedy `tlm_fifo` a mechanizmu portov, ktoré navyše pridáva metodika OVM.

7.1.2 DPI

Pre komunikáciu s hardvérovou časťou verifikačného prostredia je použité rozhranie DPI (angl. *Direct Programming Interface*). Toto rozhranie umožňuje jednoducho zahrnúť zdrojové súbory implementované v inom programovacom jazyku ako SystemVerilog do verifikačného prostredia. Rozhranie DPI je tiež použité pre pripojenie referenčného modelu k verifikačnému prostrediu.

Komunikácia medzi FPGA kartou a softvérom funguje pomocou `DMA` (angl. *Direct Memory Access*) prenosov. Tie sú naprogramované v jazyku C s využitím knižnice `libsze2`, ktorá je súčasťou platformy `NetCOPE`.

Referenčný model je automaticky generovaný nástrojmi systému Codasip vo forme zdrojových súborov v programovacom jazyku C.

7.2 Hardvérová časť

7.2.1 Platforma NetCOPE

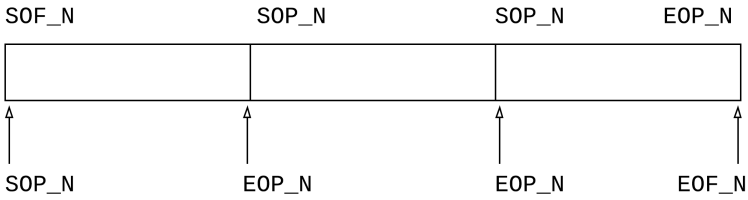
`NetCOPE` [14] je platforma určená pre rýchly a jednoduchý vývoj škálovateľných FPGA aplikácií. Aplikácie, ktoré sú akcelerované pomocou integrovaného obvodu FPGA sa väčšinou skladajú z dvoch častí: 1) hardvérová časť, ktorá sa nachádza na integrovanom obvode FPGA zahŕňajúca časovo náročné úlohy, 2) softvérová časť obsahujúca riadiace a kontrolné funkcie. Platforma `NetCOPE` vytvára v akcelerovanom prostredí rozhranie medzi softvérovou a hardvérovou časťou.

Dôležitou vlastnosťou platformy `NetCOPE` je možnosť využitia rýchlych DMA prenosov medzi počítačom a kartou FPGA.

7.2.2 Protokol Framelink

`Framelink` je protokol umožňujúci dátové prenosy, ktorý bol navrhnutý v projekte Liberouter [12]. Protokol podporuje synchronizáciu a point-to-point spojenie medzi komunikujúcimi uzlami, pričom dáta sú prenášané vo forme paketov.

Komponenty hardvérovej časti verifikačného prostredia komunikujú s využitím protokolu `Framelink`. Ten definuje na rozhraní komponentov nasledujúce signály:



Obrázok 7.1: Riadiace signály protokolu Framelink pre rámec obsahujúci 3 časti.

- `DATA` - signál slúžiaci pre prenos dát s voliteľnou šírkou (8, 16, 32, 64, 128)
- `REM` - signál, ktorý definuje počet platných bajtov v signáli `DATA`
- `SRC_RDY_N` - signál, ktorým zdroj dát dáva najavo svoju pripravenosť posielať dáta
- `DST_RDY_N` - signál, ktorým príjemca dát dáva najavo svoju pripravenosť prijímať dáta
- `SOF_N` - signál označujúci začiatok dátového rámca, ktorý môže byť zložený z viacerých častí
- `EOF_N` - signál označujúci koniec dátového rámca, ktorý môže byť zložený z viacerých častí
- `SOP_N` - signál označujúci začiatok časti, ktorá sa nachádza v dátovom rámci
- `EOP_N` - signál označujúci koniec časti, ktorá sa nachádza v dátovom rámci

Obrázok 7.1 ilustruje nastavenie riadiacich signálov pri posielaní rámca, ktorý sa skladá z troch častí.

Kapitola 8

Testovanie

Táto kapitola sa venuje testovaniu výsledkov tejto práce. Testovanie výsledného produktu zohráva významnú rolu v rámci procesu vývoja informačných technológií. Vďaka testovaniu totiž môžeme overiť kvalitu výsledného produktu. Pojem kvalita označuje splnenie požiadaviek, vhodnosť k danému účelu alebo schopnosť produktu plniť dané potreby.

Výsledkom tejto práce je verifikačné prostredie pre procesor, ktoré umožňuje akceleráciu funkčnej verifikácie. Jedná sa o akcelerovanú verziu verifikačného prostredia, ktorej návrh bol popísaný v podkapitole 6.2. Prostredie je rozdelené na dve základné časti podľa funkcie a účelu komponentov. Prvá časť má na starosti poskytovanie vstupov pre verifikovaný procesor Codix (vo forme programov). Druhá časť zabezpečuje monitorovanie výstupov procesora (obsah pamäte a registrovej zložky po vykonaní vstupného programu). Pri vývoji verifikačného prostredia, najmä hardvérových komponentov bolo potrebné samostatne testovať funkčnosť jednotlivých komponentov. Neskôr pomocou spojenia komponentov bolo možné otestovať funkčnosť väčších celkov. V závere procesu implementácie bolo možné testovať výsledok práce pri spojení všetkých komponentov verifikačného prostredia. Pri testovaní celého systému sa objavili chyby súvisiace s komunikáciou komponentov, časovaním alebo oneskorením, ktoré boli následne odstránené.

8.1 Simulácia procesora

Pre testovanie VHDL modelov sa používa technika označovaná ako testbench. Testbench [1] je taktiež kód napísaný v jazyku VHDL, ktorý tvorí nad testovaným komponentom obálku. Hlavnou úlohou testbenchu je poskytovať vstupy pre vstupné rozhranie a monitorovať výstupné rozhranie testovanej jednotky. Testovanie prebieha pomocou simulácie v prostredí programu `ModelSim` [13] , kde je možné kontrolovať priebehy signálov. Z priebehu signálov je možné pozorovať, či testovaná jednotka vykonáva svoju funkciu správne. Verifikačné prostredie implementované v rámci tejto práce bolo testované pomocou simulácie a testbenchu.

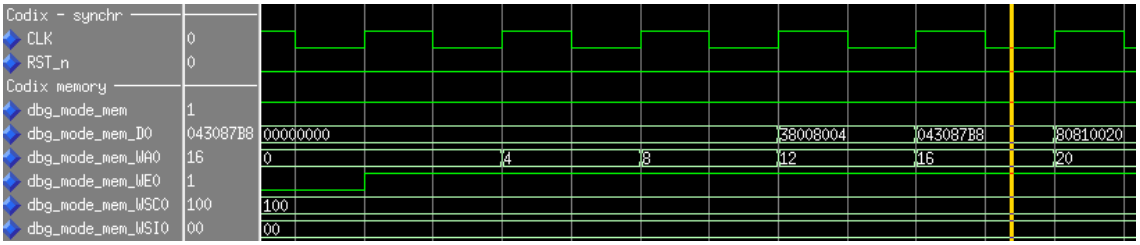
Priebeh simulácie procesora je možné podľa toku dát verifikačným prostredím rozdeliť na tieto časti:

1. zápis vstupného programu do pamäte procesora,
2. činnosť procesora,
3. čítanie obsahu pamäte, registrovej zložky a výstupného portu procesora.

Nasledujúce podkapitoly sú chronologicky usporiadané podľa vyššie uvedených základných častí simulácie. Signály, ktorých priebehy je možné vidieť na obrázkoch zo simulačného prostredia sú popísané v podkapitole 4.6.2 venovanej rozhraniu procesora Codix.

8.2 Zápis vstupného programu do pamäte procesora

Na obrázku 8.1 je uvedený začiatok komunikácie s procesorom. Komunikáciu so vstupným rozhraním zabezpečuje komponent `program_driver`. Na priebehu signálov je možné sledovať spôsob komunikácie, kedy je potrebné nastaviť očakávané hodnoty signálov na rozhraní procesora. Pri zápise dát do pamäte procesora je potrebné nastaviť signál `dbg_mode_mem` na aktívnu hodnotu a potom pomocou nástupnej hrany povoľovacieho signálu `dbg_mode_mem_WE0` sa zaháji zápis do pamäte. Zapisujú sa dáta so šírkou 32 bitov - signál `dbg_mode_mem_D0` na adresu určenú 18 bitovým signálom `dbg_mode_mem_WA0`, ktorá sa zvyšuje o hodnotu 4. Konštanty `dbg_mode_mem_WSI0` a `dbg_mode_mem_WSC0` určujú index a počet podblokov.



Obrázok 8.1: Komunikácia na vstupnom rozhraní verifikovaného procesora Codix.

8.3 Činnosť procesora

Procesor Codix zaháji svoju činnosť po tom, ako je signál `RST_n`, ktorý udržuje procesor v aktívnom resete počas práce s pamäťou, nastavený do neaktívnej hodnoty. Počas činnosti procesora je pozmenený obsah pamäte a registrovej zložky. O ukončení svojej činnosti procesor informuje pomocou aktívnej hodnoty signálu `port_halt`. Tento signál riadi činnosť monitorovacích prostriedkov na výstupnej strane procesora. Po detekcií ukončenia činnosti procesora je potrebné ho opäť udržovať v aktívnom resete počas čítania obsahu pamäte a registrovej zložky.

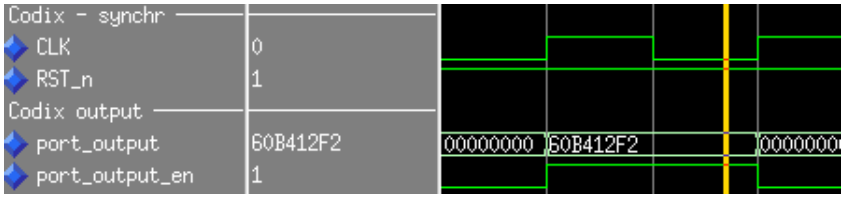
8.4 Monitorovanie výstupov procesora

Výstupom procesora po jeho činnosti je nový obsah pamäte a registrovej zložky. Naviac môže procesor počas svojej činnosti posielať hodnoty na svoj výstupný port.

8.4.1 Čítanie hodnoty na výstupnom porte

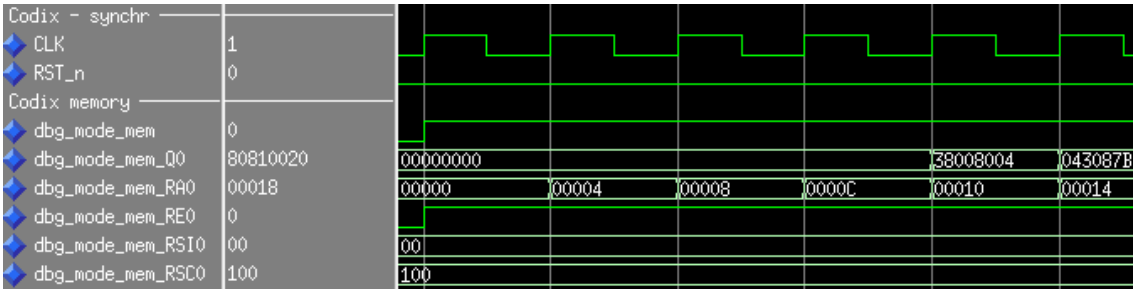
Výstupný port procesora `port_output` má šírku 32 bitov a jeho platnosť je určená pomocou signálu `port_output_en`. Na obrázku 8.2 máme možnosť vidieť príklad situácie, kedy procesor posiela na svoj výstup dáta.

Monitorovanie dát na výstupnom porte procesora má na starosti komponent `portout_monitor`.



Obrázok 8.2: Dáta na výstupnom porte procesora.

8.4.2 Čítanie obsahu pamäte

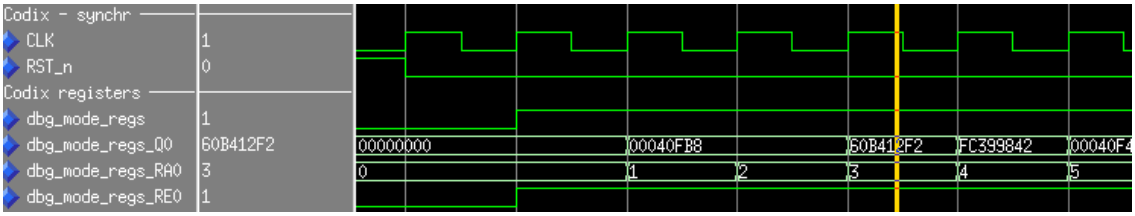


Obrázok 8.3: Čítanie obsahu pamäte procesora Codix.

Komunikácia pri čítaní dát z pämate procesora je podobná ako pri zápise. Pomocou nastavenia signálov `dbg_mode_mem` a `dbg_mode_mem_RE0` na aktívnu hodnotu sa zaháji čítanie pamäte. Pomocou inkrementácie adresy po maximálnu možnú hodnotu dôjde k vyčítaniu obsahu pamäte. Dáta sa objavujú s nástupnou hranou synchronizačných hodín na signály `dbg_mode_mem_Q0`.

Začiatok čítania obsahu pamäte môžeme vidieť na obrázku 8.3.

8.4.3 Čítanie obsahu registrovej zložky



Obrázok 8.4: Čítanie obsahu registrovej zložky procesora Codix.

Čítanie registrovej zložky je zahájené ihneď po detekcii aktívnej hodnoty signálu `port_halt`, ktorým procesor dáva najavo ukončenie svojej činnosti. Pomocou povoľovacieho signálu `dbg_mode_regs_WE0` v aktívnej hodnote a inkrementujúcej sa adresy `dbg_mode_regs_RA0` dôjde k vyčítaniu obsahu registrovej zložky.

Obrázok 8.4 dokumentuje komunikáciu medzi komponentom `register_monitor` procesorom Codix.

Kapitola 9

Záver

Cieľom tejto práce bolo navrhnúť verifikačné prostredie pre procesory podľa verifikačnej metodiky OVM a s využitím akceleračnej platformy HAVEN akcelerovať verifikáciu procesora Codix. Práca sa mala zamerať na slabé miesto, resp. nevýhodu funkčnej verifikácie, ktorou je využívanie pomalej softvérovej simulácie verifikovaného systému.

Problém pomalého behu funkčnej verifikácie pri simulácií procesora bol riešený implementáciou verifikačného prostredia v hardvéri. Do verifikačného prostredia v hardvéri sa presunul verifikovaný procesor a verifikačné komponenty potrebné pre komunikáciu s procesorom. Princíp presunu verifikovanej jednotky z prostredia softvéru do prostredia hardvéru je prevzatý z platformy HAVEN. V tejto práci bola akceleračná platforma HAVEN využitá ako základ pri implementácii verifikačných komponentov, ktorý bolo možné rozšíriť a prispôsobiť pre riešenie daného problému.

Výsledkom práce je verifikačné prostredie určené pre hardvér, pomocou ktorého je možné dosiahnuť akceleráciu funkčnej verifikácie. Výsledné verifikačné prostredie zabezpečuje komunikáciu s verifikovaným procesorom. Na začiatku do jeho pamäte nahrá vstupný program, ktorý následne procesor začne vykonávať. Po ukončení činnosti procesora je vyčítaný obsah jeho pamäte a registrovej zložky. Tento obsah sa posiela do softvérovej časti pre vyhodnotenie a porovnanie s referenčnými výsledkami.

Prepojenie softvérovej a hardvérovej časti v rámci akcelerovanej verzie verifikačného prostredia nebolo realizované. Dôvodom bola vysoká časová náročnosť pri testovaní a ladení komponentov hardvérovej časti. Bolo totiž potrebné otestovať každý komponent osobitne v rámci skupiny susediacich komponentov a tiež v rámci celého systému. Takýto zdĺhavý postup spoločne s nízkou rýchlosťou simulácie celého verifikačného prostredia v hardvéri mali za následok posuny v časovom pláne riešenia tejto práce.

Preto aby bolo možné spustiť funkčnú verifikáciu procesora s využitím hardvérovej akcelerácie je potrebné priviesť výstupy z hardvérovej časti do kontrolných mechanizmov v softvérovej časti. Vyriešenie tohoto problému ostáva výzvou pre ďalšiu prácu.

9.1 Možné rozšírenie

Ako vhodná možnosť pre rozšírenie výsledku tejto práce sa ukazuje doladenie komunikácie medzi softvérovou a hardvérovou časťou verifikačného prostredia. Väčšia efektivita pri využívaní zdrojov na integrovanom obvode FPGA by mohla byť dosiahnutá pomocou odstránenia nevyužívaných komponentov, ktoré platforma `NetCOPE` umiestňuje do hardvéru. Znovupoužiteľnosť verifikačného prostredia by mohla byť rozšírená pomocou generických

rozhraní, ktoré by zvýšili prispôsobiteľnosť verifikačného prostredia v hardvéri. Vo výsledku by teda bolo možné len pomocou zmeny týchto parametrov prispôsobiť architektúru a funkcie verifikačného prostredia pre rozličné typy procesorov.

Literatura

[1] Ashenden, P.J.: *The Student's Guide to Vhdl*. Morgan Kaufmann Series in Systems on Silicon, Morgan Kaufmann Publishers, 1998, ISBN 978-1-5586-0520-6.

[2] Iman, S.: *Step-by-Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing, 2008, ISBN 978-0-9816-5621-2.

[3] Kolektív autorov: *Codasip ® Manual*. 2011, verzia 6.4.

[4] Kolektív autorov: *CodAL Manual, reference guide*. 2012, verzia 4.2.

[5] Marcela Šimková: *Hardwarově akcelerovaná funkční verifikace*. Diplomová práce, FIT VUT v Brně, 2011.

[6] Marcela Šimková and Ondřej Lengál and Michal Kajan: HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware. *Lecture Notes in Computer Science*, ročník 2012, č. 7261, 2012: s. 247–253, ISSN 0302-9743.
URL www.fit.vutbr.cz/research/view_pub.php?id=9738

[7] Mark Glasser: *Open Verification Methodology Cookbook*. Springer, 2009, ISBN 978-1-4419-0967-1.

[8] Meyer, A.: *Principles of Functional Verification*. Elsevier Science, 2003, ISBN 978-0-0804-6994-2.

[9] Salemi, R.: *FPGA Simulation: A Complete Step-By-Step Guide*. GreatManager series, Boston Light Press, 2009, ISBN 978-0-9741-6490-8.

[10] Spear, C.: *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer London, Limited, 2008, ISBN 978-0-3877-6530-3.

[11] WWW stránky: Gleichmann Electronics Research - Products [online]. [cit. 2013-05-10].
URL www.ge-research.com/semulator.html

[12] WWW stránky: Liberouter / Cesnet TMC group [online]. [cit. 2013-05-10].
URL www.liberouter.org

[13] WWW stránky: ModelSim [online]. [cit. 2013-05-10].
URL www.model.com

[14] WWW stránky: NetCOPE FPGA platforma [online]. [cit. 2013-05-10].
URL www.invea.cz/produkty-sluzby/netcope-fpga-platforma

[15] WWW stránky: Transaction-based acceleration [online]. [cit. 2013-05-10].
URL www.cadence.com/products/sd/pages/transactionacc.aspx

[16] WWW stránky: Veloce2 - Mentor Graphics [online]. [cit. 2013-05-10].
URL www.mentor.com/products/fv/emulation-systems/veloce

[17] WWW stránky: Verification Academy [online]. [cit. 2013-05-10].
URL www.verificationacademy.com

[18] Šimková, M.; Lengál, O.; Kajan, M.: HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware. Technická zpráva, 2011.
URL www.fit.vutbr.cz/research/view_pub.php?id=9739

Příloha A

Obsah CD

Koreňový adresár disku CD obsahuje nasledujúce zložky a súbory:

`/tech_report.pdf`	– technická správa k tejto práci vo formáte pdf
`/tech_report/`	– zdrojové texty k technickej správe tejto práce
`/src/`	– zdrojové súbory tejto práce

Komponenty hardvérovej časti verifikačného prostredia sa nachádzajú v zložke:

`/src/combov2/comp/hw_ver/`

Pre každý komponent je vytvorená nasledujúca adresárová štruktúra:

`/hw_ver/komponent/sim`	– súbory pre simuláciu komponentu
`/hw_ver/komponent/synth`	– súbory pre syntézu komponentu
`/hw_ver/komponent/README`	– súbor s informáciami a návodmi

Kde `komponent` môže byť `program_driver`, `halt_monitor`, `portout_monitor`, `register_monitor`, `memory_monitor` alebo `fl_binder`.

Spustiť simuláciu pomocou programu `ModelSim` je možné nasledovne:
`vsim -do simulation.fdo` v zložke: `/hw_ver/komponent/sim/`.

Spustiť syntézu pomocou nástroja `Makefile` je možné nasledovne:
`make` v zložke: `/hw_ver/komponent/synth/`.

Kompletné verifikačné prostredie pre hardvérovú časť sa nachádza v zložke:

`/src/combov2/comp/verification_core_codix/`

Dokumentácia a návody k zdrojovým súborom sú na disku CD zahrnuté vo forme textových súborov s názvom `README`, ktoré sa nachádzajú v každej zložke, ktorá obsahuje komponent alebo celé verifikačné prostredie.