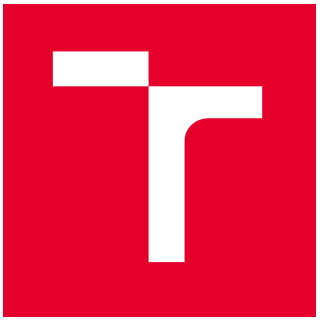


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY

DEPARTMENT OF MICROELECTRONICS

UNIFIKOVANÉ VERIFIKAČNÍ PROSTŘEDÍ DIGITÁLNÍ ČÁSTI INTEGROVANÝCH OBVODŮ SE SMÍŠENÝMI SIGNÁLY PRO AUTOMOBILOVÝ PRŮMYSL

UNIFIED VERIFICATION ENVIRONMENT FOR DIGITAL PART OF AUTOMOTIVE MIXED-SIGNAL INTEGRATED CIRCUITS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Samuel Petráš

AUTHOR

VEDOUCÍ PRÁCE — Ing. Roman Prokop, Ph.D.

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Mikroelektronika**

Ústav mikroelektroniky

Student: Bc. Samuel Petráš ***ID:*** 203317

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Unifikované verifikační prostředí digitální části integrovaných obvodů se smíšenými signály pro automobilový průmysl

POKYNY PRO VYPRACOVÁNÍ:

Popište současné metody verifikace digitální části integrovaných obvodů se smíšenými signály, jejich výhody a nevýhody. Zaměřte se také na využití hardwarové akcelerace, její přínosy, omezení a požadavky na verifikační prostředí. Navrhněte metodologii pro unifikované verifikační prostředí a jeho komponenty s možností využití hardwarové akcelerace pomocí Siemens Veloce. Do návrhu je třeba zahrnout požadavky, které na verifikaci klade automobilový průmysl.

Na základě navržené metodologie sestavte verifikační prostředí pro zvolený referenční návrh a porovnejte hardwarovou akceleraci se simulací. Zhodnoťte, pro jaké typy návrhů a testových případů je vhodné použít hardwarovou akceleraci a kdy je vhodnější klasická simulace. Zamyslete se nad možností využití hardwarové akcelerace pro menší návrhy. Doplňte navrženou metodologii o doporučení jak kombinovat hardwarovou akceleraci a simulaci pro dosažení co nejlepších výsledků.

DOPORUČENÁ LITERATURA:

[1] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[2] Verification Academy [online]. Dostupné z: https://verificationacademy.com

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 24.5.2022

Vedoucí práce: Ing. Roman Prokop, Ph.D.

Konzultant: Miloš Juhás

doc. Ing. Lukáš Fujcik, Ph.D.
předseda rady studijního programu

ABSTRAKT

Táto diplomová práca sa zaoberá problematikou unifikovaného verifikačného prostredia pre verifikáciu malých návrhov digitálnej časti integrovaných obvodov so zmiešanými signálmi. Pod pojmom unifikované verifikačné prostredie sa myslí prostredie vhodné pre simuláciu a emuláciu zároveň. V prvej kapitole sú popísané súčasné metódy verifikácie takýchto návrhov. Druhá kapitola sa venuje požiadavkám, ktoré na verifikačné prostredie implementované podľa metodológie Universal Verification Methodology (UVM) kladie emulácia a priloženej implementácii takéhoto prostredia. Tretia kapitola obsahuje praktické poznatky nadobudnuté pri implementácii unifikovaného verifikačného prostredia, problémy a ich riešenia a taktiež niekoľko porovnaní medzi simuláciou a emuláciou.

KĽÚČOVÉ SLOVÁ

verifikácia, digitálny integrovaný obvod, hardvérová akcelerácia, Siemens Veloce, emulácia, UVM, unifikované verifikačné prostredie

ABSTRACT

This thesis is concerned with unified verification environment for the verification of small designs of the digital part of integrated circuits with mixed signals. By unified verification environment is meant an environment suitable for both simulation and emulation. The first chapter describes the current verification methods of such designs. The second chapter presents the requirements that emulation places on the verification environment implemented according to the Universal Verification Methodology (UVM) and the attached implementation of proposed environment. The third chapter contains practical knowledge gained during the implementation of the unified verification environment, problems and their solutions, as well as several comparisons between simulation and emulation.

KEYWORDS

verification, digital integrated circuit, hardware acceleration, Siemens Veloce, emulation, UVM, unified verification environment

BIBLIOGRAFICKÁ CITÁCIA

PETRÁŠ, Samuel. *Unifikované verifikační prostředí digitální části integrovaných obvodů se smíšenými signály pro automobilový průmysl*. Brno, 2022. Dostupné taktiež z: https://www.vutbr.cz/studenti/zav-prace/detail/142452. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedúci práce Roman Prokop.

Prehlásenie autora o pôvodnosti diela

Meno a priezvisko študenta:	*Samuel Petráš*
VUT ID študenta:	*203317*
Typ práce:	*Diplomová práca*
Akademický rok:	*2021/22*
Téma záverečnej práce:	*Unifikované verifikační prostředí digitální části integrovaných obvodů se smíšenými signály pro automobilový průmysl*

Prehlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúceho záverečnej práce a s použitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, predovšetkým som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Zb., vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka č. 40/2009 Zb. Českej Republiky.

V Brne dňa: 23. mája 2022 ----------

podpis autora

Poďakovanie

Ďakujem konzultantovi Ing. Milošovi Juhásovi za účinnú metodickú a odbornú pomoc a ďalšie cenné rady pri spracovaní diplomovej práce. Taktiež by som chcel poďakovať vedúcemu práce Ing. Romanovi Prokopovi, Ph.D., bez ktorého ochoty by táto práca nevznikla.

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM ZDROJOVÉHO KÓDU

ÚVOD

Cieľom verifikácie je overiť súlad špecifikácie a návrhu (v tomto prípade návrhu digitálnej časti integrovaného obvodu so zmiešanými signálmi; ďalej len „návrh“). Aj napriek zvyšovaniu veľkosti a komplexnosti návrhov sa vďaka pokroku vo verifikačných metódach počet potrebných prototypov pred zahájením produkcie nezvyšuje. [1]

Prvý prototyp býva úspešný iba v tretine projektov (*first silicon success*). Najviac sa v návrhoch vyskytujú logické alebo funkčné chyby. Tieto chyby zapríčiňujú, že zhruba 70 % projektov je dokončených po očakávanom termíne. [1]

V priemere zaberá verifikácia 55 % času potrebného na dokončenie projektu. Pokiaľ sa do štatistík nezapočítajú projekty využívajúce veľké množstvo vopred verifikovaných IP (Intellectual Property) jadier, toto číslo stúpne na 60-70 %. V tomto rozsahu sa dlhodobo pohybuje väčšina projektov a vyplýva z toho, že urýchlenie verifikácie môže mať na dodržanie termínu dokončenia projektu významný vplyv. [1]

Jednou z možností, ako verifikáciu návrhu urýchliť, je použitie hardvérovej akcelerácie. Hardvérová akcelerácia sa delí na FPGA (Field-Programmable Gate Array) prototypovanie a emuláciu. Nárast popularity hardvérovej akcelerácie je zachytený v grafe na obrázku 1.4.

Táto práca sa venuje využitiu emulátorov triedy Veloce Strato od firmy Siemens EDA pre návrhy automobilového priemyslu. Konkrétne vlastnosti návrhov pre ktoré je práca cielená sú uvedené v kapitole *2.2 Implementácia*.

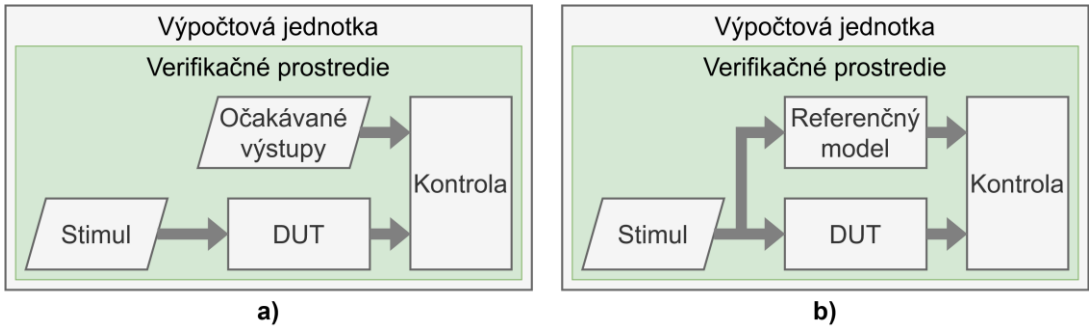
V prvej kapitole práce sú opísané súčasné metódy verifikácie návrhov a taktiež obsahuje teoretický úvod do hardvérovej akcelerácie. Druhá kapitola popisuje problematiku unifikovaného verifikačného prostredia (UVP), teda prostredia, ktoré je vhodné pre simuláciu aj emuláciu. Pre implementáciu UVP priloženú k tejto práci bola použitá verifikačná metodológia UVM (Universal Verification Methodology). Cieľom práce nie je poskytnúť všeobecný prehľad o UVM metodológii ale overiť jej využitie pre emuláciu návrhov automobilového priemyslu. Z toho dôvodu vznikol ako súčasť práce aj referenčný návrh, pre ktorý bolo UVP zostavené. V druhej kapitole je ďalej možné nájsť popis jednotlivých častí priloženého UVP, pričom najväčšia časť je venovaná modelu registrov. Ako je v tejto kapitole ukázané, implementácia modelu registrov, najmä tých, ktorých správanie je potrebné manuálne definovať, nie je triviálna. Tretia kapitola začína popisom toho, aké nástroje sú dostupné pre implementáciu UVP cielenú na emulátory triedy Veloce Strato. Následne sú v tejto kapitole rozoberané bežné problémy a požiadavky, s ktorými je možné sa stretnúť pri implementácii UVP, a predstavuje ich riešenia. Okrem toho je v tretej kapitole uvedených niekoľko porovnaní medzi simuláciou a emuláciou.

1. SÚČASNÉ METÓDY VERIFIKÁCIE DIGITÁLNYCH INTEGROVANÝCH OBVODOV

V súčasnosti existuje viacero metód ako verifikovať súlad medzi návrhom a jeho špecifikáciou. Prvá podkapitola sa venuje funkčnej simulácii. Ďalej bude predstavená formálna verifikácia, meranie pokrytia (*coverage*) a hardvérová akcelerácia. Z hľadiska tejto práce sú najpodstatnejšie kapitoly o funkčnej simulácii a hardvérovej akcelerácii, ktoré slúžia ako teoretický úvod pre druhú a tretiu kapitolu.

1.1 Funkčná simulácia

Základom funkčnej simulácie je verifikačné prostredie nazývané *testbench*. Overenie korektnosti návrhu prebieha privedením vhodného stimulu na vstupy verifikovaného návrhu (Device Under Test – DUT) a následným porovnaním výstupov DUT s očakávanými hodnotami, ktoré sú považované za korektné. Jedná sa teda o dynamickú metódu. Stimul môže byť manuálne definovaný vopred alebo vygenerovaný počas simulácie. To isté platí aj pre očakávané výstupy. Manuálne definovanie očakávaných výstupov je veľmi zdĺhavé a náchylné na chyby, preto sa vytvárajú referenčné modely, ktoré očakávané hodnoty generujú automaticky na základe stimulu (pozri obrázok 1.1). Referenčné modely sú typicky vytvorené s vyššou úrovňou abstrakcie, čo znamená, že využívajú aj nesyntetizovateľné prvky HDL (Hardware Description Language) jazykov. Prípadne môžu byť vytvorené v inom programovacom jazyku. Princípom verifikácie funkčnou simuláciou je teda zdvojenie implementácie. Správanie syntetizovateľnej implementácie je porovnávané s implementáciou na vyššej úrovni abstrakcie. Aby sa zabránilo opakovaniu chýb v oboch implementáciách (v interpretácii špecifikácie) je odporúčané, aby ich nevytvárala tá istá osoba. [2][3]



Obrázok 1.1 Funkčná simulácia – zjednodušené verifikačné prostredie

Stimul môže byť napísaný tak, aby bola verifikovaná určitá cielená časť návrhu (nazýva sa *directed*), alebo môže byť generovaný pseudonáhodne v definovaných medziach (nazýva sa *constrained-random*). Medze sa definujú tak, aby bol stimul obmedzený na podmnožinu legálnych kombinácii vstupov. Avšak aj v prípade použitia

stimulu typu *constrained-random* je pre zaujímavé (okrajové) prípady vhodné napísať časť stimulu ako *directed*. Zbieraním štatistík o pokrytí kódu a funkcionality (pozri kapitolu *1.3 Meranie pokrytia (Coverage)*) je možné odhaliť nedostatky v stimule – niektoré časti návrhu nie sú overené, alebo je niektorá časť stimulu redundantná. [2]

Pokrytie stavového priestoru funkčnou simuláciou závisí od stimulu. Porovnanie so statickou analýzou je uvedené v kapitole *1.2 Formálna verifikácia* a znázornené na obrázku 1.2.

V prípade detegovania chybného výstupu DUT je úlohou verifikačného prostredia na túto skutočnosť upozorniť výpisom v zázname o priebehu simulácie. Okrem textového záznamu o priebehu simulácie je možné počas simulácie zaznamenávať priebeh jednotlivých signálov v čase (*waveform*). Úlohou verifikačného inžiniera je nájsť pôvod detegovanej chyby. [2]

Kontrola výstupov vykonávaná každý hodinový cyklus vyžaduje, aby boli referenčné modely presné s rozlíšením jedného hodinového cyklu (*cycle-accurate*). Alternatívou je kontrola výstupov na základe transakcií, prebiehajúca s menším časovým rozlíšením (*transaction-accurate*). [3]

Spôsob, akým je generovaný vstupný stimul a akým sú monitorované výstupy z DUT, sa podľa pozorovateľnosti delí na tri úrovne [3]:

1. *Black box* – ovplyvniteľné a pozorovateľné sú len vstupné a výstupné vývody DUT. Hlavnou výhodou je jednoduchosť a nezávislosť na návrhu. S verifikáciou je tým pádom možné začať skôr ako pri ostatných úrovniach pozorovateľnosti. Avšak identifikovať prípadné miesto vzniku pozorovanej chyby môže byť v tomto prípade náročné. V niektorých prípadoch je nemožné rozhodnúť o korektnosti návrhu len na základe hodnôt na vstupných a výstupných vývodoch DUT.
2. *White box* – 100% viditeľnosť a ovládateľnosť vnútorných signálov návrhu. Plná ovládateľnosť umožňuje nastavenie vnútorných stavov podľa potreby (napríklad stav konfiguračných registrov alebo stavového automatu). Plná viditeľnosť umožňuje priame pozorovanie zmien vnútri návrhu. Vďaka tomu je jednoduchšie nájsť konkrétne miesto vzniku chyby. Pri zmene návrhu nastáva riziko že bude potrebná zmena verifikačného prostredia alebo vstupného stimulu a prípadne referenčného modelu.
3. *Grey box* – kompromis medzi oboma predchádzajúcimi prístupmi. Ovládateľnosť a viditeľnosť vnútorných signálov DUT je limitovaná. Typicky sú prístupné len rozhrania jednotlivých blokov, z ktorých sa DUT skladá. Zmena verifikačného prostredia alebo vstupného stimulu je potrebná len pokiaľ sa zmenia tieto rozhrania.

Simulátory použité pre funkčnú simuláciu sa rozdeľujú na dva typy: riadené udalosťami (*event-driven*) a riadené hodinovými cyklami (*cycle-based*). Simulátory riadené udalosťami vyhodnocujú výsledok logického výrazu len pri zmene vstupných

hodnôt. Simulátory riadené hodinovými cyklami rozdelia návrh podľa hodinových domén a výsledky logických výrazov pre jednotlivé domény vyhodnocujú hromadne pri každej zmene hodinového signálu. Asynchrónne udalosti je teda možné simulovať len v simulátore typu *event-driven*. Rýchlosť simulácie záleží od počtu vyhodnocovaných výrazov RTL (Register Transfer Level) popisu alebo hradiel v prípade simulácie popisu na hradlovej úrovni (*netlist*). Vzhľadom na to, že pri hardvérovej akcelerácii je verifikovaný návrh (DUT) implementovaný paralelne v programovateľnej logike, degradácia rýchlosti pri náraste veľkosti návrhu je výrazne nižšia ako v prípade softvérovej simulácie. [3][4]

Pred spustením simulácie prebieha statická kontrola kódu, odhaľujúca chyby v návrhu bez použitia stimulu. Môže sa jednať napríklad o signál bez budiča alebo nesúlad v šírke zberníc. [2]

1.2 Formálna verifikácia

Jedná sa o matematickú metódu verifikácie návrhu. Jej výsledkom je definitívny dôkaz korektnosti, alebo nekorektnosti, určitej vlastnosti návrhu, ktorej platnosť je požadované overiť pre ľubovoľné vstupy (ďalej len „vlastnosť“). Každá vlastnosť musí byť popísaná. Najčastejšie sa na ich definíciu používajú prvky SystemVerilog Assertions (SVA) popísané v štandarde jazyka SystemVerilog (štandard IEEE 1800). Ako však bude opísané nižšie, niektoré nástroje formálnej verifikácie dokážu tieto vlastnosti definovať automaticky. [5]

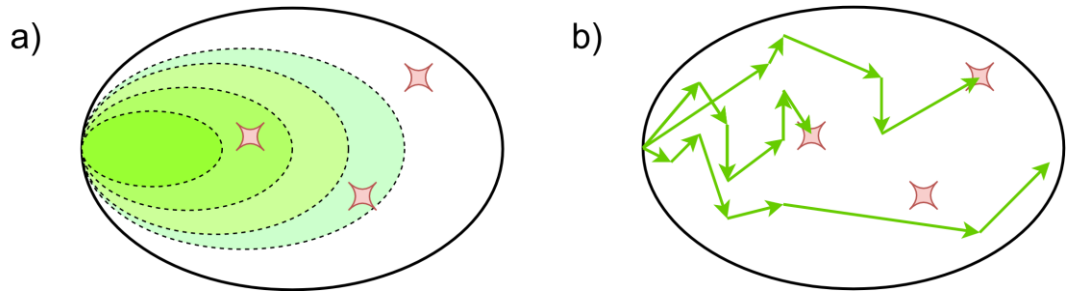
Na rozdiel od ostatných verifikačných metód rozoberaných v tejto kapitole sa jedná o statickú metódu. To znamená, že návrh nepotrebuje stimul ani verifikačné prostredie. Pre jej spustenie v tom najjednoduchšom prípade stačí len RTL alebo popis návrhu na hradlovej úrovni (pozri tabuľky 1.1 a 1.2). Z toho vyplýva jedna z jej výhod – je možné použiť ju v raných fázach projektu, pričom cieľom je nájdenie čo najväčšieho počtu chýb v čo najkratšom čase (tzv. *bug hunting*). V neskorších fázach projektu sa používa na zvýšenie spoľahlivosti ako doplnok iných metód verifikácie. [5]

Formálna analýza vychádza z počiatočného stavu návrhu. Počiatočný stav môže byť prípadne extrahovaný z ľubovoľného bodu simulácie. Na začiatku analýzy sa identifikujú prvky držiace stav. Pre každý takýto prvok sa vytvorí logická funkcia nasledujúceho stavu na základe aktuálneho stavu a vstupov. V každom kroku formálnej analýzy sa podľa týchto logických funkcií určia dosiahnuteľné nasledujúce stavy. Tento proces je konečný, avšak matematický problém s hĺbkou analýzy exponenciálne narastá a vzniká tým problém nazývaný *State Explosion Problem*. Možným riešením je zníženie počtu analyzovaných stavov. To sa dá dosiahnuť napríklad zjednodušením definovaných vlastností, zanedbaním niektorých vstupov (typicky testovacieho režimu prevádzky), znížením šírky zberníc alebo zredukovaním niektorých modulov len na vstupy a výstupy (*black box*). [5]

Rozlišujeme štyri rôzne výsledky formálnej analýzy, pričom výsledky číslo tri a štyri sú následkom spomínaného problému *State Explosion Problem* [6]:

1. Našli sa všetky unikátne stavy a definované vlastnosti boli splnené.
2. Nájde sa porušenie jednej z definovaných vlastností. V takom prípade poskytne nástroj postup, ktorým dosiahol porušenie. Tento postup sa nazýva protipríklad (anglicky *counterexample* – CEX) a je to dôkaz nekorektnosti danej vlastnosti.
3. Systému dôjde voľná pamäť a nie je schopný pokračovať v analýze.
4. Analýza sa stane príliš komplexná a nachádzanie nových stavov trvá neprakticky dlhú dobu. V takom prípade sa všetky pokryté stavy považujú za plne verifikované, a všetky nepokryté za neverifikované. Vlastnosti, ktorých korektnosť nebola dokázaná a nenašiel sa pre ne ani protipríklad, sa nazývajú nerozhodné. Pre rozhodnutie o ich korektnosti je vhodné zvážiť inú formu verifikácie alebo zníženie počtu analyzovaných stavov.

Rozdiel medzi formálnou verifikáciou a funkčnou simuláciou je znázornený na obrázku 1.2. Biely ovál predstavuje celý stavový priestor návrhu a červenou farbou sú symbolizované chyby v návrhu. Formálna verifikácia postupne rozširuje pokrytie verifikovaných stavov, až kým analýza neskončí jednou zo štyroch možností uvedených vyššie. Funkčná simulácia postupuje na základe definovaného stimulu, vďaka čomu sa môže ľahšie dostať do hlbších stavov, ale pokryje len veľmi malú podmnožinu stavového priestoru. [6]



Obrázok 1.2 Pokrytie stavov a) formálna verifikácia; b) funkčná simulácia [6]

Nároky na definovanie vlastností závisia od konkrétneho nástroja na formálnu verifikáciu. Firma Siemens EDA ponúka viacero takýchto nástrojov v portfóliu Questa Formal Solutions. Jednotlivé nástroje sú zamerané na rôzne účely a štádiá návrhu digitálneho integrovaného obvodu. V riešení špecifických problémov sú efektívnejšie ako zvyšné metódy verifikácie, pretože vytvorenie stimulu pre overenie niektorých vlastností môže byť časovo náročné alebo prakticky nemožné. Podľa náročnosti na použitie sa delia do dvoch skupín: Formal Apps a Model Checking. Pre všetky nástroje z oboch skupín platí, že po dokončení formálnej analýzy je možné zobraziť výsledky v interaktívnej forme pomocou užívateľského rozhrania a taktiež ich uložiť do UCDB (Unified Coverage Database) databázy, vďaka čomu môžu byť zlúčené s ostatnými

výsledkami verifikácie. Zo stĺpca vstupov v tabuľkách 1.1 a 1.2 nižšie je zrejmé, že väčšina nástrojov vyžaduje viac ako len popis návrhu na RTL alebo hradlovej úrovni.

1.2.1 Questa Formal Apps

Nástroje spadajúce do kategórie Formal Apps sú jednoduché na použitie. Pred spustením nie je potrebné definovať žiadne vlastnosti návrhu manuálne v zdrojových súboroch (napríklad použitím SVA); definície prebiehajú plne automaticky na základe dodaných súborov. Tieto súbory však nemusia byť súčasťou špecifikácie – v takom prípade je nutné počítať s časovou náročnosťou ich implementácie.

V prehľadovej tabuľke 1.1 sú uvedené funkcie a potrebné vstupy pre jednotlivé nástroje.

Tabuľka 1.1 Nástroje Questa Formal Apps [7]

Nástroj	Funkcia	Vstupy
AutoCheck	Odhaľuje časté chyby v návrhu[1]	RTL/*netlist* návrhu
SLEC[2]	Porovnáva RTL špecifikáciu a návrh	RTL/*netlist* návrhu, RTL špecifikácia
Register Check	Kontroluje správanie registrov na základe mapy registrov[3]	RTL/*netlist* návrhu, CSV/IP XACT mapa registrov
CoverCheck	Identifikuje nedosiahnuteľné stavy a poskytuje príklad, ako vo funkčnej simulácii dosiahnuť nepokryté dosiahnuteľné stavy	RTL/*netlist* návrhu, prípadne UCDB databáza
X-Check	Identifikuje propagáciu logickej hodnoty „X“	RTL/*netlist* návrhu, inicializačná sekvencia
Connectivity Check	Kontroluje prepojenia medzi blokmi na základe špecifikácie prepojení	RTL/*netlist* návrhu, CSV/XML špecifikácia prepojení
SecureCheck	Overuje prístup k citlivým úložným prvkom a signálom na základe ich špecifikácie	RTL/*netlist* návrhu, špecifikácia citlivých prvkov a signálov

1 – Napríklad: mŕtvy kód, uviaznutie stavového automatu (*deadlock*), kombinačné slučky
2 – Sequential Logic Equivalence Check
3 – Pre každý register obsahuje: adresa a veľkosť, prístupové práva, inicializačné hodnoty

1.2.2 Questa Model Checking

V prípade použitia funkcií nástroja PropCheck z kategórie Model Checking sa od užívateľa vyžaduje, aby manuálne definoval vlastnosti návrhu v kóde (napríklad použitím SVA). Z toho vyplývajú vyššie požiadavky na znalosti používateľa ako v prípade nástrojov z kategórie Formal Apps.

V prehľadovej tabuľke 1.2 sú uvedené tri rôzne funkcie, pre ktoré je možné nástroj PropCheck použiť.

Tabuľka 1.2 Funkcie nástroja Questa PropCheck [7]

Nástroj	Funkcia	Vstupy
PropCheck	Overenie správania návrhu na základe užívateľom definovaných vlastností	RTL/*netlist* návrhu, definície vlastností[1]
	Overenie správania modifikovaných IP jadier na základe formálnych knižníc pre dané IP jadrá	RTL/*netlist* návrhu, formálna knižnica daného IP jadra
	Overenie správania prototypov integrovaných obvodov spoluprácou s hardvérovou verifikačnou platformou	Prototyp integrovaného obvodu, definície vlastností[1]

1 – Môžu byť popísané napríklad použitím SVA

1.3 Meranie pokrytia (*Coverage*)

Meranie pokrytia je užitočné pre získanie predstavy o postupe a úplnosti verifikácie. Úplnosťou sa v tomto prípade myslí to, aká percentuálna časť návrhu (RTL popisu alebo funkcionality) bola počas verifikácie aktivovaná. [8]

Dve základné otázky, na ktoré meranie pokrytia odpovedá sú [8]:

- Ostáva v návrhu kód, ktorý ešte nebol pri verifikácii aktivovaný?
- Bola overená celá funkcionalita definovaná v pláne testov?

Po osvojení napomáha s odhadom časovej náročnosti verifikácie, s jej optimalizáciou a poskytuje odpovede na komplexnejšie otázky, ako napríklad [8]:

- Bola verifikovaná funkcionalita X v tom istom čase ako funkcionalita Y?
- Zasekol sa postup verifikácie z nejakého neočakávaného dôvodu?
- Je možné niektoré testy vynechať, aby sa urýchlil verifikačný proces a stále bolo dosiahnuté požadované pokrytie?

Pri meraní pokrytia je dôležitá ovládateľnosť a pozorovateľnosť. Ovládateľnosťou sa myslí možnosť ovplyvniť vnútorné správanie návrhu pomocou jeho vstupných portov. Pozorovateľnosťou sa myslí možnosť sledovať následky vnútorného správania návrhu a detegovať chyby. Verifikačné prostredie má obmedzenú schopnosť pozorovať, pokiaľ sleduje iba výstupné porty návrhu. Môže nastať situácia, kedy verifikačné prostredie vygeneruje stimul potrebný na aktivovanie určitej chyby, avšak nevygeneruje stimul potrebný na propagáciu chyby na miesto, kde je schopný túto chybu detegovať. [8]

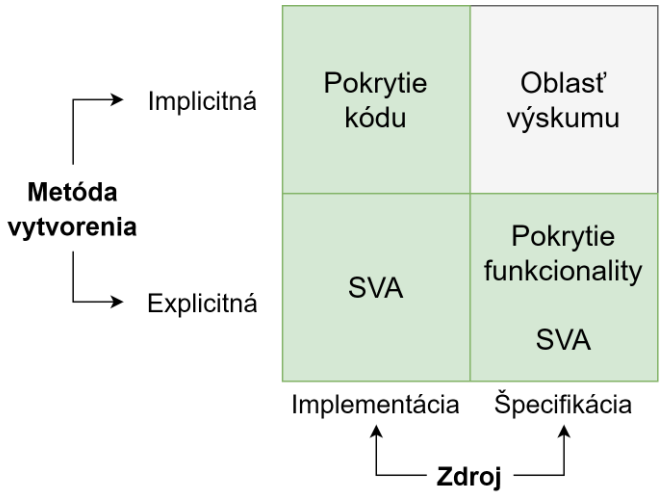
Aby bolo možné zachytiť chybu v návrhu verifikáciou, ktorá používa verifikačné prostredie, je nutné splniť nasledovné podmienky [8]:

- Musia byť vygenerované vstupy, ktoré aktivujú danú chybu
- Verifikačné prostredie musí mať schopnosť detegovať danú chybu
- Musia byť vygenerované vstupy, ktoré propagujú danú chybu na miesto, kde je možné ju detegovať

Miesta, kde dokáže verifikačné prostredie detegovať chyby, sa líšia od implementácie verifikačného prostredia. Detegovať chyby je možné napríklad na výstupoch návrhu (vysokoúrovňová pozorovateľnosť) alebo vnútri návrhu (napríklad pomocou SVA – nízkoúrovňová pozorovateľnosť). [8]

Rozlišujú sa dve metódy merania pokrytia, ktoré sa navzájom dopĺňajú, pretože ani jedna z nich nie je dostačujúca samostatne. Je možné dosiahnuť 100% pokrytie kódu, to však ešte neznamená, že bola pokrytá celá funkcionalita návrhu uvedená v špecifikácii. Vysvetlením je to, že pokrytie kódu nesleduje paralelné interakcie medzi jednotlivými časťami návrhu, ani sekvenčné správanie v čase. A naopak, je možné pokryť celú funkcionalitu návrhu bez toho, aby bol pokrytý celý kód. To by však znamenalo, že sa v návrhu nachádza funkcionalita (kód), ktorá nie je definovaná v špecifikácii, a teda sa s ňou nepočíta pri meraní pokrytia funkcionality. [8]

Pokrytie je zároveň možné rozdeliť podľa metódy vytvorenia (implicitná alebo explicitná) a zdroja (špecifikácia alebo návrh). Takéto rozdelenie je znázornené na nasledujúcom obrázku 1.3.



Obrázok 1.3 Rozdelenie merania pokrytia podľa metódy vytvorenia a zdroja [8]

Spolu tvorí pokrytie kódu a funkcionality takzvaný priestor pokrytia (alebo model pokrytia). Explicitný priestor pokrytia vytvorený na základe špecifikácie vzniká manuálnou prácou verifikačného inžiniera. Môže byť taktiež vytvorený manuálnym popisom kontrol správania návrhovým inžinierom (napríklad použitím SVA). Implicitný priestor pokrytia vzniká automatickým extrahovaním metrík z návrhu. Implicitný priestor pokrytia extrahovaný zo špecifikácie je záujmom výskumu. [8]

1.3.1 Pokrytie kódu (*Code Coverage*)

Jedná sa o implicitnú metódu merania pokrytia. Spustenie tohto typu merania pokrytia nekladie žiadne požiadavky na verifikačné prostredie a RTL návrh – spúšťa sa nastavením simulátora. Cieľom je identifikovať časti kódu, ktoré neboli doposiaľ

verifikáciou aktivované. S meraním pokrytia kódu sa odporúča začať, až keď sa návrh blíži ku koncu, a kód sa mení len minimálne. Nič však nebráni tomu spustiť meranie pokrytia kódu v ľubovoľnom bode projektu, keď už je k dispozícií RTL popis návrhu. Analýzou výsledkov je možné identifikovať príčiny nedostatkov v pokrytí kódu. [8]

Rozlišujú sa tri rôzne príčiny [8]:

- Chýba vstupný stimul potrebný na dosiahnutie nepokrytého kódu
- Chyba v návrhu alebo verifikačnom prostredí, ktorá zabraňuje dosiahnutiu určitých častí kódu
- Časti kódu môžu byť očakávane nepokryté (napríklad v danej konfigurácii IP jadra alebo kód špeciálneho režimu pri normálnom režime prevádzky návrhu)

V prípade očakávane nevyužitého kódu poskytujú analyzačné nástroje možnosť túto skutočnosť zohľadniť pri vyhodnocovaní miery pokrytia.

Toggle Coverage

Pokrytie zmien zaznamenáva počet zmien logickej hodnoty jednotlivých bitov každého signálu. Požiadavkou projektu môže byť napríklad aspoň jedna zmena z logickej úrovne 0 do logickej úrovne 1 a naopak. [8]

Line Coverage

Pokrytie riadkov sleduje, ktoré riadky kódu boli počas verifikácie vykonané. Pre každý riadok počíta, koľko krát sa vykonal. Často odhaľuje ťažko dosiahnuteľné vetvy podmienečných štruktúr, ktoré vyžadujú stimul, aký nebol počas verifikácie dodaný. Niektoré vetvy môžu byť nedosiahnuteľné kvôli chybám v návrhu alebo kvôli výskytu nevyužitého kódu. [8]

Statement Coverage

Pokrytie príkazov sleduje, koľko krát boli počas verifikácie vykonané jednotlivé príkazy. Je považovaný za užitočnejší ukazovateľ pokrytia ako *Line Coverage*, pretože na jednom riadku sa môže nachádzať viacero príkazov, alebo môže byť jeden príkaz rozdelený na viac riadkov kódu. [8]

Block Coverage

Pokrytie blokov kódu identifikuje či sa určitý blok kódu vykonal, alebo nie. Blok kódu je definovaný ako kód vykonávaný iba po splnení určitej podmienky alebo kód vnútri definície metódy. Pre blok kódu platí, že pokiaľ sa vykoná jeho prvý riadok, vykonajú sa všetky. [8]

Branch Coverage

Pokrytie vetiev vyhodnocuje, či boli vykonané všetky vetvy štruktúr, ktoré rozhodujú medzi vykonávaním blokov kódu na základe logickej funkcie (napríklad konštrukty *if*, *case*, *while*, *repeat*, *forever*, *for* a *loop*). Zložitosť logickej funkcie nie je v tomto prípade podstatná, prihliada sa iba na výsledok (logická pravda, nepravda). [8]

Expression Coverage

Logický výraz sa považuje za pokrytý len ak sa počas verifikácie vyskytnú všetky kombinácie vstupov. Za vstup sa v tomto prípade považuje každá časť výrazu, ktorú je možné vyhodnotiť ako logickú pravdu alebo nepravdu (napríklad $(a > b)$ sa považuje za jeden vstup a výraz $x = ((a > b) \,\&\, (c < d))$ má teda dva vstupy). [9]

Focused Expression Coverage (FEC)

FEC sa zameriava na vplyv jednotlivých vstupov na výstup logického výrazu. Vplyv jednotlivých vstupov sa považuje za pokrytý len ak dokáže kontrolovať výstup za podmienky, že ostatné vstupy sú v takom stave, aby nemali na zmenu výstupu vplyv. Každý vstup musí byť schopný zmeniť výstup výrazu do oboch logických hodnôt. [9]

Finite-State Machine (FSM) Coverage

Moderné nástroje na meranie pokrytia sú schopné v kóde identifikovať štruktúry stavových automatov. To umožňuje vyhodnocovať štatistiky, ako napríklad koľko krát sa stavový automat nachádzal v ktorom stave a koľko krát nastal prechod medzi jednotlivými susednými stavmi. [8]

1.3.2 Pokrytie funkcionality (*Functional Coverage*)

Jedná sa o explicitnú metódu merania pokrytia. Umožňuje získať prehľad o tom, ktorá funkcionalita návrhu definovaná v špecifikácii už bola implementovaná a verifikovaná bez nutnosti manuálnej kontroly výstupov verifikácie (*requirements tracing*). To sa s výhodou využíva napríklad pri simulácii s náhodným stimulom (*constrained-random stimulus*). [8]

Aby mohlo meranie pokrytia funkcionality fungovať, je najskôr nutné vytvoriť priestor pokrytia. To prebieha v dvoch krokoch [8]:

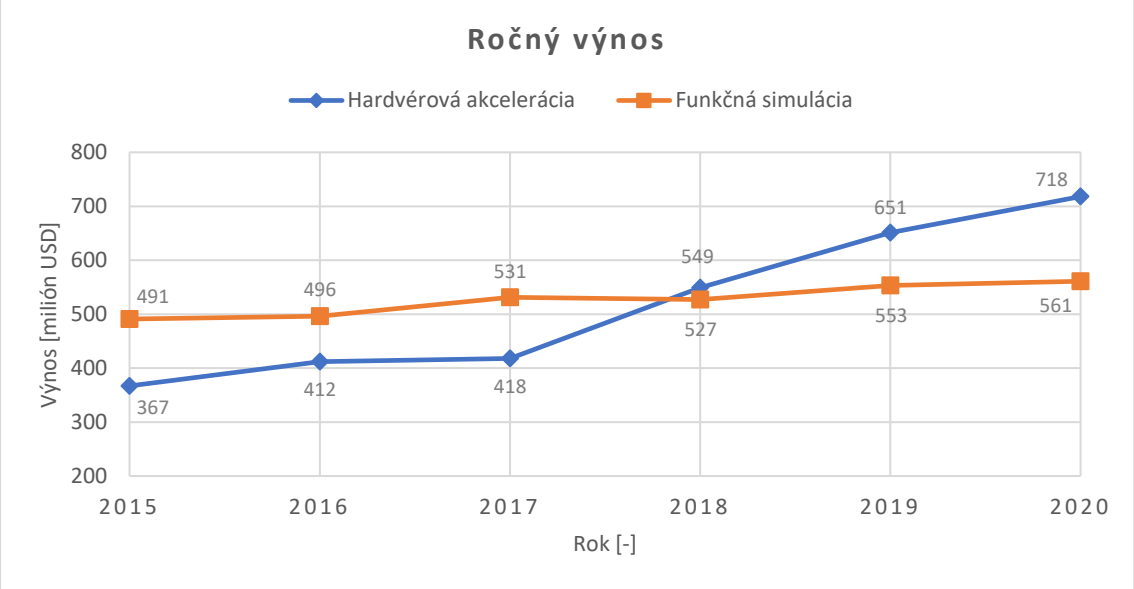
- Identifikuje sa funkcionalita, ktorej pokrytie je požadované merať
- Implementujú sa nástroje na meranie každej takejto funkcionality (napríklad použitím konštruktov jazyka SystemVerilog *covergroup* a *coverpoint*)

S meraním pokrytia funkcionality sledovaním jednotlivých signálov sa odporúča začať, rovnako ako pri meraní pokrytia kódu, až keď sa návrh blíži ku koncu a kód sa mení len minimálne. V prípade použitia identifikátorov funkcionality v špecifikácii (*requirements tracing*), je meranie pokrytia nezávislé na stave návrhu.

Na rozdiel od merania pokrytia kódu je okrem samotného RTL návrhu v tomto prípade potrebné manuálne implementovať nástroje na meranie pokrytia. [8]

1.4 Hardvérová akcelerácia

Rozlišujú sa dve metódy hardvérovej akcelerácie – FPGA prototypovanie a emulácia. V súčasnosti sa obe metódy využívajú ako doplnok simulácie, pričom každá má svoje výhody a nevýhody. Nárast popularity hardvérovej akcelerácie v posledných rokoch je zrejmý z grafu na obrázku 1.4.



Obrázok 1.4 Ročný výnos: hardvérová akcelerácia a funkčná simulácia [10]

Obecnou výhodou hardvérovej akcelerácie oproti simulácii je jej rýchlosť, najmä pri verifikácii veľkých návrhov a návrhov so softvérom. Dnešné platformy pre hardvérovú akceleráciu podporujú návrhy s veľkosťou v miliardách ASIC ekvivalentných hradiel (anglicky *billion gates* – BG). [11]

Hardvérová akcelerácia umožňuje pripojenie návrhu k reálnym cieľovým zariadeniam (napríklad SSD riadič k reálnemu SSD) a otestovať ho s reálnym stimulom. Tento typ verifikačného prostredia sa anglicky nazýva *in-circuit emulation* (ICE). Prináša to so sebou však aj určité nevýhody. Rýchlosť hardvérovej akcelerácie často nedosahuje rýchlosť cieľových zariadení a na ich spojenie je preto potrebné použiť vyrovnávaciu pamäť. Takáto vyrovnávacia pamäť je závislá od cieľového zariadenia a preto je nutné ju vždy vhodne prispôsobiť, prípadne manuálne vymeniť (bežne aj stovky vodičov [12]). Trasovanie chýb v návrhu je vzhľadom na nedeterministické reálne prostredie v prípade ICE mimoriadne náročné. Práve kvôli týmto nevýhodám vznikol nový typ verifikačného prostredia pre hardvérovú akceleráciu – virtuálne verifikačné prostredie, kde sa namiesto

reálnych cieľových zariadení používajú softvérové modely s vysokou úrovňou abstrakcie. [11]

Vzhľadom na to, že je návrh implementovaný v reálnom hardvéri, pracuje na rozdiel od simulácie iba s dvojstavovou logikou (s logickými hodnotami „1“ a „0“). [2]

1.4.1 FPGA prototypovanie

FPGA čipy nie sú vhodné na implementáciu návrhov, ktoré vyžadujú kapacitu dosiahnuteľnú len prepojením viacerých FPGA čipov. Rovnako nebývajú navrhnuté tak, aby umožnili viditeľnosť do návrhu – je potrebné kompilovať monitorovacie sondy, ktoré zaberajú miesto. Dôvod, prečo sa FPGA prototypovanie používa, je rýchlosť vykonávania návrhu, ktorá dosahuje rádovo 10 až 100-násobok rýchlosti emulácie. [11]

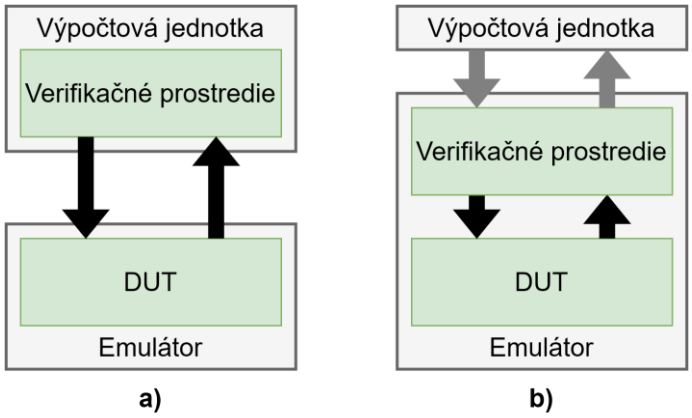
1.4.2 Emulácia

Emulácia pristupuje k hardvérovej akcelerácii z opačného konca – zameriava sa na 100% viditeľnosť, možnosti ladenia (*debugging*) a bezproblémovú implementáciu návrhu do veľkého počtu čipov za cenu nižšej rýchlosti (nízke jednotky MHz). V súčasnosti sa rýchlosť emulátorov nijak významne nezvyšuje, rastie však ich kapacita. Namiesto komerčných FPGA čipov používajú vlastné čipy s prispôsobenou architektúrou. [11][13][14]

Virtuálne verifikačné prostredie sa realizuje podľa obrázka 1.5a. Emulovaný je len návrh (DUT) s modelmi, ktoré konvertujú transakčné príkazy z vysokej úrovne abstrakcie na signálové sekvencie (protokol), a naopak. Tieto modely sa nazývajú BFM (Bus Functional Model). Zvyšné časti verifikačného prostredia sú vykonávané vo výpočtovej jednotke pripojenej k emulátoru (typicky výpočtový server; nazýva sa *co-model* [15]). Vďaka komunikácii na transakčnej úrovni dokáže *co-model* dosiahnuť rýchlosť emulátora. [11][16]

V určitých testových prípadoch alebo pri optimalizovanom verifikačnom prostredí sa môže stať, že sa komunikácia medzi výpočtovou jednotkou a emulátorom stane najviac časovo náročným elementom. V takom prípade je pre dosiahnutie vyššej rýchlosti vhodné zvážiť presunutie verifikačného prostredia do emulátora, ako je možné vidieť na obrázku 1.5b. Komunikácia medzi výpočtovou jednotkou a emulátorom tak bude minimálna (napríklad informácie o stave emulácie). Tento prístup však vyžaduje aby bolo celé verifikačné prostredie syntetizovateľné. [11][16]

Použitím virtuálneho verifikačného prostredia sa zachová pomer hodinových signálov návrhu a modelu cieľového zariadenia – nebude teda potrebná vyrovnávacia pamäť. V kombinácii so 100% viditeľnosťou to umožňuje analýze spotreby dosiahnuť výsledky s presnosťou v jednotkách percent, čo je oproti simulácii rádové zlepšenie. Spotrebu návrhu ovplyvňuje aj softvér a vďaka rýchlosti emulácie je možné analyzovať spotrebu pri realistickej záťaži – napríklad načítanie operačného systému. V simulácii by takáto analýza mohla trvať neprakticky dlho. [11][17]



Obrázok 1.5 Verifikačné prostredie pre emuláciu [16]

Spojením emulácie a FPGA prototypovania cez virtuálne verifikačné prostredie je možné dosiahnuť optimálnu kombináciu rýchlosti a ladiacich možností. Návrh sa môže vykonávať v FPGA až kým sa nenarazí na chybu; v tom momente sa verifikácia presunie do emulátora, kde prebieha ladenie. Umožňuje to začať s verifikáciou softvéru prakticky zároveň s verifikáciou hardvéru. [11]

1.4.3 Siemens Veloce Strato

Firma Siemens EDA ponúka v portfóliu Veloce hardvér pre emuláciu a FPGA prototypovanie. Aktuálna generácia emulátorov sa nazýva Veloce Strato a zahŕňa 5 modelov.

Tabuľka 1.3 Prehľad modelov emulátorov generácie Veloce Strato [14]

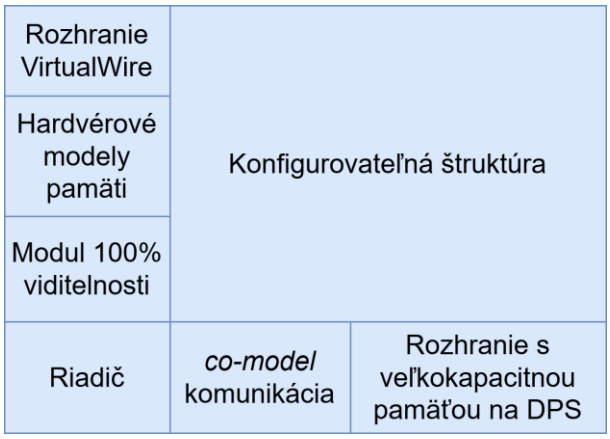
Model	Kapacita v BG	Topológia			Prepojené moduly
		Kabinety	Moduly	AVB[1]	
M	2,5	2	4	64	áno
Mi	4-krát 0,64	2	4	64	nie
T	1,25	1	2	32	áno
Ti	2-krát 0,64	1	2	32	nie
TiL	0,64	1	1	16	-

1 – Advanced Verification Board – kapacita 0,04 BG [13]

Maximálny počet užívateľov, ktorý môžu emulátor využívať súčasne sa rovná počtu AVB (za predpokladu, že žiadny z užívateľov nepotrebuje väčšiu kapacitu ako je kapacita AVB). Modely generácie Veloce Strato je možné modulárne kombinovať a tým navýšiť kapacitu pre emuláciu podľa potreby. Najvyššia podporovaná kapacita je 15 BG (vznikne spojením šiestich modelov Veloce Strato M). Modely, ktoré nemajú prepojené jednotlivé moduly (pozri tabuľku 1.3), dokážu emulovať návrhy len do veľkosti kapacity jedného modulu (16 AVB). [13]

Čip Crystal3

Generácia Veloce Strato využíva na emuláciu čip Crystal3 s architektúrou zobrazenou na obrázku 1.6. Čip má programovateľnú logiku, nie je to však FPGA. Je optimalizovaný pre rýchlu a bezproblémovú syntézu. Vysokorýchlostné spojenie nazývané VirtualWire zabezpečuje presun dát medzi jednotlivými čipmi; emulovaný návrh tak môže zaberať kapacitu stoviek čipov Crystal3. Čipy sú spojené priamo, pre dosiahnutie čo najnižšej latencie, ale aj nepriamo pomocou programovateľných spojení pre vyššiu flexibilitu pri emulovaní veľkých návrhov. [13]



Obrázok 1.6 Architektúra čipu Crystal3 [13]

Aby bola implementácia BFM modelov syntetizovateľná, pri ich písaní je potrebné dodržiavať určité pravidlá, ktoré budú popísané v kapitole *3 Emulácia prakticky*. Čip Crystal3 nie je optimalizovaný pre finálnu implementáciu, ale pre emuláciu. Čip taktiež podporuje optimalizované modelovanie pamätí. Ako však bude vysvetlené v kapitole *2.2.2 Model pamäti EEPROM*, modely, ktoré sú k dispozícií, nemusia byť pre niektoré návrhy použiteľné. [13]

Pre zachytávanie hodnôt signálov v čase má čip Crystal3 dedikovanú časť a tým pádom nespomaľuje emuláciu, ani neznižuje kapacitu programovateľnej logiky. [13]

Operačný systém Veloce Strato

Operačný systém Veloce Strato využíva plnú viditeľnosť do návrhu na ladenie pomocou prístupu k ľubovoľným signálom, ukladanie priebehu signálov v čase (*waveform*), analýzu spotreby alebo meranie pokrytia. Medzi pokročilé funkcie patrí napríklad podpora pre prerušenie emulácie na určitom riadku RTL popisu návrhu (*breakpoint*), uloženie a následné obnovenie stavu emulácie, ďalej zaznamenanie priebehu a následné prehranie emulácie a povoľovanie SVA pre vybrané behy emulácie. Tieto funkcie je možné dosiahnuť len vďaka prispôsobenému hardvéru. [13]

Operačný systém Veloce Strato je spätne kompatibilný aj so staršou generáciou Veloce2 Maximus. [18]

Veloce Strato hardvér

Hardvér je navrhnutý tak, aby zabezpečil efektívne rozdelenie zdrojov pre viacero užívateľov. Umiestňuje sa do dátových centier a je nepretržite dostupný cez internet. [13]

Kompilácia návrhu prebieha na virtuálny hardvér (AVB) a pri spustení sa implementuje do ktorýchkoľvek voľných AVB. [13]

Pripájanie reálnych cieľových zariadení v režime ICE je realizované použitím prepínacieho systému, ktorý pripojené zariadenia presmeruje na aktuálne využité AVB. Hardvér podporuje aj kombinovanie virtuálneho a ICE stimulu. [13]

2. UNIFIKOVANÉ VERIFIKAČNÉ PROSTREDIE

Pre implementáciu UVP bol zvolený jazyk SystemVerilog (štandard IEEE 1800) a metodológia Universal Verification Methodology (štandard IEEE 1800.2). Jedná sa o najpoužívanejší jazyk a metodológiu (oboje boli použité zhruba v 75 % projektoch v roku 2020 [1]).

Od UVP sú očakávané nasledovné vlastnosti [19]:

- Zameniteľnosť – musí byť vhodné pre hardvérovú akceleráciu (v tomto prípade emuláciu) a zároveň simuláciu
- Flexibilita – musia byť zachované výhody plynúce z použitia metodológie UVM a jazyka SystemVerilog (medzi výhody patrí napríklad behaviorálne modelovanie, objektovo orientované programovanie a znovupoužiteľnosť)
- Výkon – aby sa emulácia ekonomicky oplatila, musí priniesť výrazné zrýchlenie voči simulácii a zároveň nesmie simuláciu negatívne ovplyvniť zmenami vo verifikačnom prostredí

Nutnou požiadavkou na verifikačné prostredie určené pre emuláciu je rozdelenie hierarchie na dve domény: 1) HVL doména (názov vychádza zo skratky pre Hardware Verification Language) – vykonáva sa vo výpočtovej jednotke (*co-model*) a 2) HDL doména (názov vychádza zo skratky pre Hardware Description Language) – vykonáva sa v emulátore. Vzhľadom na hierarchickú štruktúru UVM je rozdelenie na HVL a HDL domény potrebné urobiť vo vrstve transaktorov (patrí tam napríklad *monitor* a *driver*). Takéto rozdelenie je kompatibilné so zvyškom verifikačného prostredia. [19][20]

Z toho pre UVP vyplývajú nasledovné požiadavky [20]:

- Kód HVL a HDL domény musí byť kompletne oddelený (rozdielne súbory, rozdielne hierarchie)
- UVM komponenty zasahujúce do oboch domén (komponenty vo vrstve transaktorov) musia byť rozdelené na dve časti. Časť rozdeleného transaktora v HDL doméne sa nazýva BFM model a časť v HVL doméne *proxy* trieda (zástupná trieda). Tieto dve časti spolu komunikujú na transakčnej úrovni. Nemôžu medzi nimi prechádzať signály ani hierarchické referencie. Viac o tomto rozdelení je uvedené v kapitole *2.1 Rozdelenie vrstvy transaktorov (BFM-proxy pár)*.
- Na posuv simulačného času z HVL domény nie je možné použiť konštrukty jazyka SystemVerilog *wait*, # alebo @. Časový posuv je potrebné nahradiť synchronizovaním sa podľa hodinového signálu v HDL doméne (pozri kapitolu *3.4 Posuv simulačného času*).
- Komunikácia medzi HVL a HDL doménou by pre dosiahnutie čo najvyššej rýchlosti emulácie mala byť minimálna

HVL doména nie je syntetizovaná a preto obsahuje všetky potrebné vysokoúrovňové konštrukty ako napríklad triedy (na ktorých je metodológia UVM postavená). Naopak

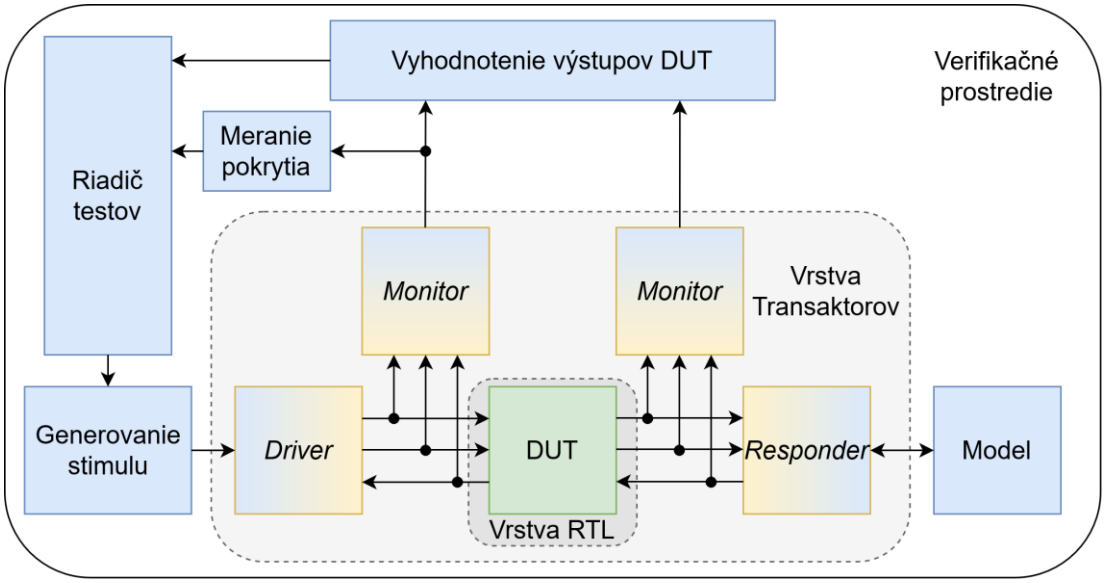
všetko v HDL doméne musí byť syntetizovateľné a synchrónne s hodinovým signálom. Simulátory triedy Veloce Strato podporujú pre jednoduchšie modelovanie syntézu niektorých bežne nesyntetizovateľných konštruktov (pozri kapitolu *3.1 Direktíva pre syntézu XRTL nadstavby*). [20]

Pre dosiahnutie čo najvyššieho zrýchlenia simulácie použitím emulácie je potrebné presunúť všetky komponenty verifikačného prostredia, komunikujúce s DUT na signálovej úrovni, do emulátora. Verifikačné prostredie vykonávané v simulátore tak pracuje len s vysokoúrovňovými dátami a s emulovanou časťou komunikuje cez transakcie (pozri kapitolu *3.3 Transakcie v emulácii*). [19]

Rozhodnutie, či implementovať verifikačné prostredie ako unifikované, je dôležité urobiť hneď na počiatku projektu. V opačnom prípade by to mohlo znamenať potrebu prerobiť už napísané komponenty. [21]

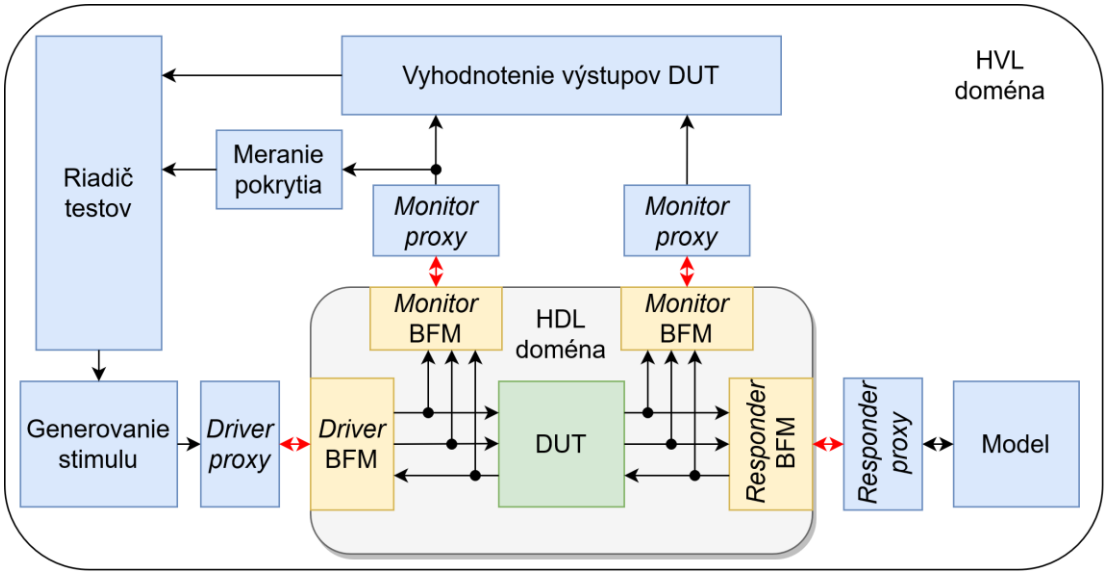
2.1 Rozdelenie vrstvy transaktorov (BFM-*proxy* pár)

V UVM metodológii je transaktor štandardne tvorený kombinovaným kódom pre transakčnú komunikáciu s verifikačným prostredím a signálovú komunikáciu s DUT. Implementuje sa ako *proxy* trieda spojená s BFM modelom (blok *interface*) pomocou ukazovateľa typu *virtual interface*. Prístup k signálom DUT je cez tento ukazovateľ možný priamo z *proxy* triedy. Príklad takejto štruktúry je zobrazený na obrázku 2.1. Emulátory triedy Veloce Strato podporujú spojenie HVL a HDL domény pomocou ukazovateľa typu *virtual interface*, avšak prístup k signálom DUT musí byť realizovaný nepriamo, volaním užívateľom definovaných metód (*function* alebo *task*) v BFM modeli. Komunikácia volaním metód cez ukazovateľ je možná aj v opačnom smere – z HDL domény do HVL domény. Používa sa na to ukazovateľ na *proxy* triedu (patriacu k danému BFM) nazývaný *back-pointer*. Využíva sa napríklad pri odosielaní transakcií na analýzu do HVL domény. Transakcie medzi emulátorom a výpočtovou jednotkou sú obmedzené v tom, aké dátové typy môžu prenášať. Viac o transakciách v emulátore je uvedené v kapitole *3.3 Transakcie v emulácii*. [19][20][22]



Obrázok 2.1 Štandardná štruktúra UVM [23]

Štruktúra UVM upravená pre emuláciu je zobrazená na obrázku 2.2. Červené šípky znázorňujú transakčnú komunikáciu medzi HVL a HDL doménou. Všetko v šedej časti obrázka 2.2 bude vykonávané v emulátore. [20]



Obrázok 2.2 Štruktúra UVM upravená pre emuláciu (Unifikované Verifikačné Prostredie) [23]

Vybrané časti zdrojového kódu uvedené v prílohe *B* demonštrujú ako je takéto rozdelenie možné implementovať v prípade transaktora *monitor* a *driver*.

2.2 Implementácia

Hlavnými charakteristikami návrhov, pre ktoré je implementované UVP cielené sú: interakcia s analógovou časťou integrovaného obvodu – spracovanie asynchrónnych signálov, testabilita, použitie pamäti EEPROM a veľkosť do 100 tisíc ASIC ekvivalentných hradiel.

Ako súčasť práce vznikla implementácia UVP a referenčný návrh. Ich hierarchie je možné nájsť v prílohe *A*. Referenčný návrh predstavuje budič H-mostíku inšpirovaný integrovaným obvodom DRV8705 od firmy Texas Instruments. Okrem funkcionality H-mostíka je v návrhu implementované SPI (Serial Peripheral Interface) rozhranie so sedemdesiatimi ôsmimi registrovými poľami, prevodník s postupnou aproximáciou (Successive Approximation – SAR), kontrolný obvod *watchdog* a pamäť EEPROM. V nasledujúcich podkapitolách sú uvedené informácie k implementácii modelu registrov a pamäti EEPROM. Návrh digitálnej časti integrovaného obvodu so zmiešanými signálmi sa taktiež nezaobíde bez implementácie resynchronizácie asynchrónnych vstupov a obvodu pre potlačenie zákmitov. Asynchrónnemu stimulu sa venuje kapitola *3.6 Asynchrónny* .

Pre implementáciu UVP bola použitá verzia UVM metodológie 1.1d dodávaná k simulátoru Questa 2019.3. Konkrétna verzia simulátora bola zvolená z dôvodu kompatibility s verziou emulátora Veloce Strato TiL v20.0.2.

2.2.1 Model registrov

Neoddeliteľnou súčasťou návrhu digitálnych častí integrovaných obvodov sú registre. Ich využitie je rôznorodé. Môžu byť použité napríklad na konfiguráciu správania, ukladanie výsledkov alebo hlásení o stave integrovaného obvodu. Súčasťou UVM metodológie sú prostriedky pre uľahčenie ich verifikácie. V nasledujúcom texte je uvedených niekoľko relevantných poznatkov nadobudnutých pri implementácii modelu registrov pre referenčný návrh.

Nástroj na generáciu modelu registrov

Pri implementácii UVP postaveného na UVM metodológii sa ako užitočný osvedčil program Register Assistant UVM. Konkrétne bola použitá verzia 2019.3, ktorá je súčasťou inštalácie simulátora Questa 2019.3. Tento program slúži na generovanie hierarchie modelu registrov, ktorá môže byť v závislosti na návrhu pomerne rozmerná a zdĺhavá pri manuálnej definícii. Vygenerovaná knižnica (*package*) pre referenčný návrh má vyše 1400 riadkov. Vstupné dáta sú akceptované vo formáte CSV (Comma-Separated Values) a výstupom je SystemVerilog knižnica s definíciami potrebných tried. Program vykoná počas generovania niekoľko kontrol konzistencie vstupných dát (napríklad prekrytie adries alebo úplnosť povinných údajov). Vybrané registre z použitého vstupného CSV súboru sú uvedené v tabuľke 2.1. [24]

Program akceptuje oveľa väčšie množstvo vstupných parametrov, avšak pre generovanie hierarchie modelu registrov, použitej v implementovanom UVP, boli stĺpce v tabuľke 2.1 postačujúce. Za zmienku stojí automatické generovanie merania pokrytia pre jednotlivé registre a registrové polia. Adresa registrov (stĺpec *Register Address* v tabuľke 2.1) bola špecifikovaná, aby mohla byť použitá funkcia *Auto-Instancing* (prepínač *autoinstancing*). Táto funkcia slúži na automatické inštancovanie registrov v bloku registrov na základe uvedenej adresy – bez potreby špecifikovať registrovú mapu alebo blok registrov. Názov cieľového bloku registrov sa uvádza použitím prepínača *block*. [24]

Tabuľka 2.1 Časť vstupných dát pre program Register Assistant UVM

Register Width	Register Address	Register Name	Field Name	Field Offset	Field Width	Field Access	Field Reset Value	Field is Reserved
8	0x01	st_mem	err_ee	0	1	RO[1]	0x00	
8	0x01	st_mem	rsvd	1	7	RO	0x00	TRUE
8	0x03	st_opc	err_uv	0	1	W1C[2]	0x00	
8	0x03	st_opc	err_ov	1	1	W1C	0x00	
8	0x03	st_opc	err_tsd	2	1	W1C	0x00	
8	0x03	st_opc	rsvd	3	5	RO	0x00	TRUE
8	0x42	rwl_cfg_1	tsd_thr	0	8	RW[3]	0x00	

1 – *Read-Only*; registrové pole je cez SPI rozhranie možné iba čítať

2 – *Write 1 to Clear*; registrové pole je cez SPI rozhranie možné resetovať zapísaním 1 do daného poľa

3 – *Read-Write*; registrové pole je cez SPI rozhranie možné čítať aj zapisovať

Vzhľadom na to, že správanie niektorých registrov muselo byť upravené použitím spätného volania (*callback*), vygenerovaný blok registrov má názov s príponou *_base* (používa sa na označenie tried, ktoré budú neskôr rozšírené). Tento blok registrov deklaruje a nakonfiguruje všetky registre, potom vytvorí mapu registrov a vloží do nej jednotlivé registre na príslušné adresy.

Úprava správania modelu registrov metódami spätného volania (*callback*)

V UVM metodológii existujú dva mechanizmy na úpravu správania modelu registrov – hákovanie (*hook*) a spätné volanie (*callback*). Ich správanie sa líši v závislosti od spôsobu prístupu k registrom alebo registrovým poliam. Rozlišujú sa dva typy prístupu – aktívny a pasívny. Za aktívny prístup sa považuje každé volanie prístupových metód modelu registrov, ktoré vyvolá transakciu cez príslušného *agenta* (napríklad metódy *write* a *read*). Pasívny prístup k modelu registrov neprechádza cez *agenta* priradeného k daným registrom – zmeny hodnôt teda nie sú prenesené do DUT. [24]

Príkladom pasívneho prístupu je implementácia predikcie obsahu registrov s výsledkami SAR prevodníka na základe pozorovania signálov v DUT (pozri zdrojový kód triedy *mtrd_scb_predictor*, konkrétne implementáciu metódy *write_sar*).

Hákovanie odkazuje na prázdne virtuálne metódy *pre_read*, *post_read*, *pre_write* a *post_write* v triede registra (*uvm_reg*) a registrového poľa (*uvm_reg_field*). Implementáciou týchto metód v odvodených triedach je možné upraviť správanie registra pri aktívnom prístupe. V tom však spočíva ich nevýhoda – nie sú volané pri pasívnom prístupe. [24]

Pasívny prístup sa implementuje volaním metódy *predict* v reakcii na zaznamenané transakcie. Aby bolo možné upraviť správanie registrových polí aj pri pasívnom prístupe, je potrebné implementovať virtuálnu metódu spätného volania *post_predict*, definovanú v triede *uvm_reg_cbs*. [24]

Zo štandardu IEEE 1800.2 je možné vyčítať informácie zhrnuté v tabuľke 2.2. Tieto informácie sú kritické pre úspešnú implementáciu metód spätného volania a pasívneho prístupu.

Tabuľka 2.2 Správanie metódy predict v závislosti na parametroch kind a path

Parameter *kind* UVM_PREDICT_*	**Parameter *path* UVM_***	**Metóda *post_predict* bude zavolaná**	**Prístupové pravidlá budú rešpektované**
DIRECT	**-**	Nie	Nie
READ/WRITE	**BACKDOOR**	Áno	Nie
	FRONTDOOR	Áno	Áno
	PREDICT[1]	Áno	Áno

1 – Parameter *path* bude mať hodnotu UVM_PREDICT v prípade, že transakcia prejde cez prediktor

Metódy spätného volania sú implementované rozšírením triedy *uvm_reg_cbs* (pozri súbor so zdrojovým kódom *regmodel_cbs.sv*, ktorý obsahuje všetky triedy implementujúce rôzne metódy spätného volania pre jednotlivé druhy registrov). V priloženom UVP boli implementované triedy obsahujúce metódy spätného volania pre nasledovné správanie registrových polí:

- Trieda *rwl_field_cb*: registrové pole s voľným prístupom pre čítanie a zápisom povoleným iba v prípade, že iné špecifikované registrové pole nadobudne definovanú hodnotu
- Trieda *st_src_field_cb*: registrové pole, ktorého hodnota, prípadne spolu s hodnotou *n* ďalších registrových polí, určuje hodnotu cieľového stavového registra – jeho hodnota je výsledok logickej operácie *OR* všetkých *n + 1* registrových polí
- Trieda *math_op_src_field_cb*: registrové pole, ktoré je jedno z polí určujúcich hodnotu stavového registra, pričom hodnota stavového registra je určená rovnicou *(adc <= (thr + offset)) ? 1 : 0* alebo *(adc >= (thr - offset)) ? 1 : 0*, kde hodnota registrových polí znamená nasledovné: *adc* – výsledok AD

prevodníka; *thr* – prahová hodnota, pri ktorej sa mení hodnota stavového registra; *offset* – posuv voči prahovej hodnote (typicky môže ísť o stavový register signalizujúci kritické hodnoty napätia alebo teploty)

- Trieda *math_op_dst_field_cb*: stavové registrové pole, ktorého hodnota je určená tromi ďalšími registrovými poľami opísanými v bode vyššie – toto spätné volanie je potrebné, pretože daný stavový register bol vygenerovaný s prístupom W1C (zápis hodnoty 1 resetuje daný bit), ale jeho resetovanie musí byť podmienené uvoľnením podmienky nastavenia aktívnej hodnoty (výsledok matematického vzorca uvedeného v bode vyššie musí byť 0)

Triedy s implementovanými metódami spätného volania je potrebné skonštruovať a priradiť metódou *uvm_reg_field_cb:add* k registrovým poliam, pre ktoré sa majú volať [25]. Jednotlivým poliam je možné priradiť viacero takýchto tried, pričom poradie, v ktorom budú volané, je určené postupnosťou, v akej sú k registrovým poliam priradené, a parametrom *ordering* metódy *uvm_reg_field_cb:add* [25]. Tento mechanizmus bol využitý na priradenie tried *math_op_dst_field_cb* a *st_src_field_cb*, práve v tomto poradí, relevantným stavovým registrovým poliam, aby bolo zaručené, že najskôr sa skontroluje, či je možné zmeniť hodnotu daného registrového poľa, a až následne sa vyhodnotí, či sa zmení aj hodnota cieľového stavového registrového poľa, na ktoré má ukazovateľ metóda spätného volania v triede *st_src_field_cb*. Takéto priradenie a konštruovanie prebieha v triede *mtrd_spi_reg_block*, ktorá rozširuje vyššie spomenutú základnú triedu, vygenerovanú programom Register Assistant UVM s príponou *_base*.

Finálnej implementácii metódy spätného volania v triede *st_src_field_cb* (a obdobne aj v triede *math_op_src_field_cb*) predchádzal pokus o iný spôsob aktualizácie hodnoty v stavovom registrovom poli. Tento neúspešný pokus zlyhal, pretože metóda *post_predict* je v skutočnosti volaná ešte pred samotným zápisom aktualizovanej hodnoty do registrového poľa. Preto nie je možné získať jeho aktualizovanú hodnotu metódou *get_mirrored_value* pred dokončením metódy *predict*. V praxi to znamená, že registrové pole, ktoré chce aktualizovať hodnotu stavového registrového poľa, musí mať ukazovatele na všetky ostatné registrové polia, od ktorých závisí hodnota stavového registrového poľa.

Volatilné registre

Za volatilné sa považujú všetky registre, ktorých hodnotu nie je možné predikovať len na základe transakcii na komunikačnom rozhraní – v prípade priloženého referenčného návrhu ide o SPI rozhranie [26]. Inak povedané – hodnoty, ktoré nie je možné predikovať len na základe operácii s registrovým modelom (napríklad *read* alebo *write*). Môže to byť napríklad register s výsledkom AD prevodu alebo stavový register. Správanie niektorých z volatilných registrov je možné modelovať pomocou metód spätného volania (opísané v predchádzajúcich odstavcoch). Iné však môžu vyžadovať predikovanie implementované v prediktore *scoreboard* triedy, buď na základe stimulu zo sekvencií

alebo sledovaním signálov priamo v DUT. Nasledujúca citovaná práca sa venuje využitiu SVA práve za účelom predikovania volatilných registrov [27].

Pasívne predikovanie hodnoty v registroch s výsledkami SAR prevodníka, implementované v triede *mtrd_scb_predictor*, potrebuje po zápise vyvolať metódu *post_predict* aby sa mohli aktualizovať hodnoty v stavových registroch. Zároveň nesmie rešpektovať prístupové pravidlá, aby mohla byť zapísaná hodnota do registra s prístupom iba pre čítanie (RO). Z toho dôvodu bola zvolená kombinácia parametrov *kind* = UVM_PREDICT_WRITE a *path* = UVM_BACKDOOR.

Hierarchia

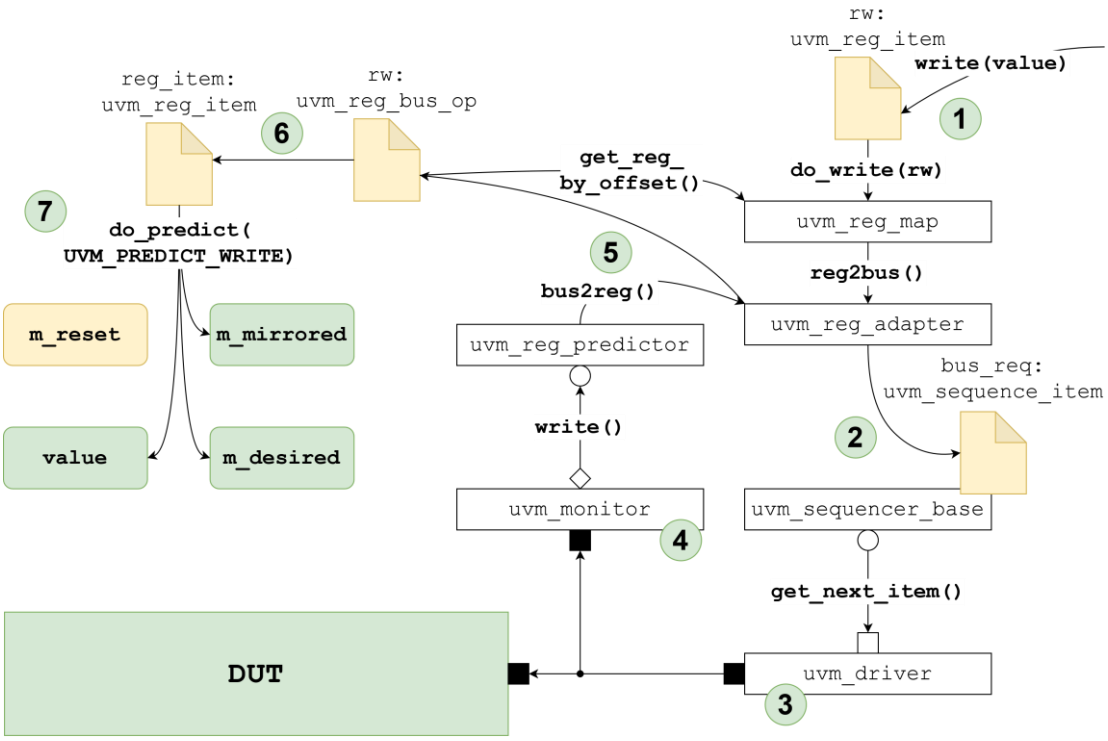
Vo verifikačnom prostredí sa inštancuje vždy trieda odvodená od základnej triedy *uvm_reg_block*, ktorá je najvyššie v hierarchii registrových blokov. V tomto prípade by to mohla byť priamo trieda *mtrd_spi_reg_block*, avšak pre prípad, že by v budúcnosti mohlo byť UVP rozšírené o ďalšie registrové bloky, bol vytvorený registrový blok *mtrd_top_reg_block*. Tento blok nerobí nič viac, len inštancuje registrový blok SPI registrov a priradí mapu tohto bloku do svojej mapy *default_map*. Vďaka tomu sa bude použitie mapy bloku *mtrd_top_reg_block* rovnať použitiu mapy bloku SPI registrov.

Tento registrový blok je skonštruovaný v hierarchicky najvyššom prostredí (trieda odvodená od *uvm_env*) a následne je predaný ukazovateľ do všetkých relevantných objektov nižšie v hierarchii.

Prevod volaných metód modelu registrov na transakcie

Po zavolaní jednej z metód pre zápis alebo čítanie registrov v DUT je potrebné previesť toto volanie na transakciu, ktorá môže byť prenesená medzi *proxy* triedou a BFM modelom a následne previesť zachytenú transakciu na objekt, ktorým je možné aktualizovať model registrov. Napríklad pri volaní metódy *write* sa vykoná nasledujúca sekvencia [28]:

1. Vytvorí sa objekt *uvm_reg_item*, ktorý obsahuje informácie o zápise
2. Adaptér v prostredí s relevantným transaktorom prevedie objekt *uvm_reg_item* na príslušný transakčný objekt
3. *Driver* vykoná transakciu
4. *Monitor* túto transakciu zachytí a pošle ju do prediktora (nemýliť si s prediktorom v objekte *scoreboard*; v tomto prípade ide o prediktor v prostredí s relevantným transaktorom)
5. Prediktor požiada adaptér o konverziu transakčného objektu na registrovú operáciu
6. Registrová operácia sa prevedie na objekt *uvm_reg_item*
7. Výsledný objekt *uvm_reg_item* sa použije na aktualizáciu dát v modeli registrov



Obrázok 2.3 Postupnosť metódy write (model registrov) [28]

Adaptér a prediktor spomínaný vyššie sú v priloženom UVP inštanciované iba v prostredí *mtrd_spi_env*. Jedná sa o jediný transaktor, ktorý je v tomto prostredí implementovaný s modelom registrov.

2.2.2 Model pamäti EEPROM

Emulátory triedy Veloce Strato ponúkajú hardvérové prostriedky pre modelovanie pamätí vo veľkostiach od 1 KB až po 1 GB [29]. Tieto modely sú cielené pre pamäte typu RAM s pevnou veľkosťou slova 8, 32 alebo 64 bitov [29]. Pre modelovanie firemných EEPROM IP jadier nie sú vhodné, pretože tie majú často komplikovanejšie rozhrania a menšie veľkosti s rôznorodou šírkou slova. Navyše ich modely bývajú doplnené o testabilitu a kontroly parametrov, ktoré nie je možné syntetizovať.

Pre emuláciu je teda potrebné vytvoriť model so zhodným rozhraním, ktorý splní funkciu pamäti EEPROM počas emulácie (napríklad inicializácia konfiguračných registrov). Verifikácia pamäti EEPROM bude vzhľadom na uvedené nedostatky hardvérových modelov prebiehať v simulácii.

Pamäť EEPROM je v priloženom UVP modelovaná ako registrové pole so stavovým automatom pre čítanie obsahu. Okrem čítania na signálovej úrovni sú v modeli implementované metódy pre okamžitý prístup k registrovému poľu. Takáto implementácia je kompatibilná s emulátormi triedy Veloce Strato, pričom v prípade potreby by mohla byť rozšírená napríklad o zápis alebo iné operácie špecifické pre konkréte IP jadro.

Tento model slúži na inicializáciu konfiguračných SPI registrov po resetovaní DUT. Inicializácia samotného modelu pamäti EEPROM prebieha z HVL domény, volaním metódy v BFM modeli (trieda *std_mem_acc_bfm*), ktorý následne volá metódu okamžitého prístupu v modeli EEPROM pamäti. BFM model má prístup k registrovému poľu cez referenciu, ktorá mu je predaná v module *hdl_top*. Hierarchický prístup cez referencie funguje vďaka 100% viditeľnosti aj v emulátoroch triedy Veloce Strato, nie len v simulácii. V HVL doméne sú metódy na prácu s EEPROM modelom implementované v konfiguračnej triede (*std_mem_acc_config*), rovnako ako v prípade iných prostredí.

Okrem inicializácie z textového súboru obsahujúceho dáta v hexadecimálnom formáte (metóda *init_from_hex*) sú implementované aj iné metódy pre okamžitý prístup k registrovému poľu v HDL doméne. Alternatívou implementáciou by mohlo byť presunutie inicializačných metód do HDL domény, keďže emulátory triedy Veloce Strato podporujú systémovú metódu *$fopen*.

3. EMULÁCIA PRAKTICKY

Všetky simulácie boli realizované s procesorom Intel Xeon Gold 6154 (3,00 GHz) a všetky emulácie na verzii emulátoru Siemens Veloce Strato TiL v20.0.2 s výpočtovou jednotkou (*co-model)* s procesorom Intel Xeon Gold 6246 (3,30 GHz).

3.1 Direktíva pre syntézu XRTL nadstavby

Syntetizovateľná podmnožina jazyka SystemVerilog sa nazýva RTL. Emulátory triedy Veloce Strato podporujú aj jej nadstavbu nazývanú eXtended RTL (XRTL). Táto nadstavba umožňuje syntetizovať bežne nesyntetizovateľné prvky, ako napríklad behaviorálne generovanie hodinového signálu, implicitné stavové automaty, bloky *initial* a *final* alebo metódy implementujúce transakčnú komunikáciu medzi doménami. Keďže sú XRTL prvky popisované bežnými konštruktmi, ktoré sú súčasťou štandardu jazyka SystemVerilog, aj tento kód môže byť simulovaný na ktoromkoľvek simulátore, ktorý podporuje štandard IEEE 1800. [20][23]

Využitie XRTL v emulácii musí byť vždy označené príslušnou direktívou. Syntax direktívy nie je súčasťou jazyka SystemVerilog – je daná použitým kompilátorom a v kóde sa uvádza ako komentár. Kompletný zoznam direktív, ktoré emulátory triedy Veloce Strato podporujú, je možné nájsť v príručke k emulátoru [22].

3.1.1 Generovanie hodinového a resetovacieho signálu

Hodinový a resetovací signál je možné generovať v *initial* bloku s nasledovnou direktívou [22].

```
// tbx clkgen
```

Zdrojový kód 3.1 Direktíva – generátor hodinového alebo resetovacieho signálu

Jedná sa o jediný prípad kedy je v HDL doméne povolené behaviorálne modelovanie (posuv simulačného času pomocou konštruktu #). Je možné nastaviť frekvenciu, fázový posuv voči ostatným hodinovým alebo resetovacím signálom a striedu. Pri ich písaní je potrebné dodržiavať prísne pravidlá uvedené v manuáli k emulátoru. V nasledujúcom príklade zdrojového kódu je uvedená jedna z možností, ako napísať generovanie hodinového signálu s premennými parametrami *phase, low*, a *high*. [22]

```
1: // tbx clkgen
2: initial begin
3:   clk = 0;
4:   #(phase)
5:   forever begin
6:     #(low)  clk = 1;
7:     #(high) clk = 0;
8:   end
9: end
```

Zdrojový kód 3.2 Generovanie hodinového signálu

Odporúčaný dátový typ premenných použitých pre definovanie fázového posuvu (*phase*) a striedy (*low* a *high*) je *shortint* (16 bitov) [22]. Pri syntéze s dátovým typom *int* (32 bitov) sa vygeneruje upozornenie TBXC-15526, ktoré nesie informáciu o tom, že takáto veľkosť premennej môže negatívne ovplyvniť výkon. Pri testovacích behoch emulácie s dátovým typom *int* nebol zaznamenaný nárast doby behu oproti dátovému typu *shortint*. Maximálny rozmer, ktorý syntéza akceptuje pre tieto premenné je 48 bitov.

V prípade, že celý návrh a všetky BFM modely (teda celá HDL doména) reagujú iba na jednu hranu generovaného hodinového signálu, je možné použiť optimalizáciu, pri ktorej bude zvolená hrana neaktívna. Keďže emulátor spracováva všetky udalosti iba na nábežnej hrane svojho hodinového signálu, nazývaného *uclock*, bude to znamenať, že emulátor zosynchronizuje tieto hodinové signály tak, aby nábežná hrana signálu *uclock* sedela s aktívnou hranou generovaného hodinového signálu. Vo výsledku teda bude generovať užívateľský hodinový signál na plnej rýchlosti signálu *uclock* – dvojnásobok oproti prípadu, keď sú aktívne obe hrany generovaného hodinového signálu. Vzhľadom na nárast v rýchlosti emulácie, ktorý z toho plynie, je žiadúce využiť túto optimalizáciu, pokiaľ je to možné. [22]

```
// tbx clkgen inactive_negedge

// tbx clkgen inactive_posedge
```

Zdrojový kód 3.3 Direktíva – neaktívna hrana hodinového signálu

Keďže rýchlosť emulácie je priamo ovplyvnená počtom spracovávaných časových bodov, UVP s viacerými generovanými hodinovými signálmi môže využiť optimalizáciu na zníženie rozlíšenia hodinových signálov (*Clock Fidelity Reduction* – CFR), ktorá spojí viacero blízkych aktívnych hrán do jedného časového bodu a vykoná tak všetky načasované udalosti pre tieto hrany naraz. [22]

Generovanie resetovacieho signálu, ktorý bude uvoľnený po uplynutí 50 jednotiek času, by mohlo vyzerať nasledovne [22].

```
1: // tbx clkgen
2: initial begin
3:       rst_b = 0;
4:   #50 rst_b = 1;
5: end
```

Zdrojový kód 3.4 Generovanie resetovacieho signálu

3.1.2 Direktíva XRTL modulu

Každý modul, ktorý obsahuje XRTL konštrukty, musí byť označený direktívou [22].

```
// pragma attribute partition_module_xrtl
```

Zdrojový kód 3.5 Direktíva – XRTL modul

Táto direktíva sa uvádza na prvom riadku tela modulu a jej dosah pôsobnosti je len daný modul, v ktorom je uvedená – neplatí pre hierarchiu inštanciovanú v ňom. Pokiaľ modul obsahuje akúkoľvek *tbx* direktívu ale nebol označený ako XRTL modul, pri syntéze sa automaticky „povýši“ na XRTL modul. Alternatívnou možnosťou je použitie relevantného prepínača pri syntéze. [22]

3.1.3 Direktíva XRTL rozhrania (*interface*)

XRTL rozhranie, ku ktorému je možné pristupovať z HVL domény sa nazýva rozhranie transaktora (*transactor interface* – XIF). Aby syntéza takéto rozhranie rozpoznala, je ho potrebné označiť direktívou na prvom riadku tela. Následne je možné v tomto rozhraní implementovať metódy na komunikáciu medzi doménami. [22]

```
// pragma attribute <názov rozhrania> partition_interface_xif
```

Zdrojový kód 3.6 Direktíva – XRTL rozhranie (interface)

3.1.4 Direktíva metód XRTL rozhrania

Metódy definované v rozhraní transaktora (XIF), ktoré je možné volať z HVL domény použitím ukazovateľa na toto rozhranie, sa nazývajú transakčné metódy (*transactor function* alebo *transactor task* – XTF). V HDL doméne ich je potrebné označiť direktívou. [22]

```
// pragma tbx xtf
```

Zdrojový kód 3.7 Direktíva – metóda XRTL rozhrania

Platia pritom nasledujúce obmedzenia [22]:

- XTF funkcia sa musí vykonať v nulovom simulačnom čase
- XTF *task* môže využívať posuv simulačného času – v takom prípade musí byť prvý príkaz čakanie na hranu hodinového signálu (konštrukt @) a následne je v jeho tele povolené používanie len tej istej hrany daného hodinového signálu
- XTF metóda nesmie byť volaná z HDL domény
- XTF metóda musí mať všetky argumenty niektorého z dátových typov patriacich do kategórie *packed* (pozri kapitolu *3.3 Transakcie v emulácii*)
- XTF metóda nesmie mať argumenty typu *inout* alebo *ref* – povolené sú len typy *input* a *output*

3.1.5 Prístup k ukazovateľom na BFM modely (XIF)

Predanie ukazovateľov na BFM modely (XIF) z HDL domény do HVL domény je možné uskutočniť cez UVM konfiguračnú databázu v špeciálnom *initial* bloku. Tento blok nie je syntetizovaný (aj keď je implementovaný v HDL doméne – konkrétne v module *hdl_top*) a slúži len na predanie ukazovateľov. [22]

```
// tbx vif_binding_block
initial begin
  import uvm_pkg::uvm_config_db;
  ...
end
```

Zdrojový kód 3.8 Direktíva – prístup k ukazovateľom na BFM modely

Je potrebné označiť ho príslušnou direktívou a dodržať nasledujúce podmienky [22]:

- Musí v ňom byť importovaná konfiguračná databáza UVM metodológie (za predpokladu použitia UVM metodológie)
- Okrem importovania konfiguračnej databázy a volania statických funkcii v ňom nesmie byť nič iné (v prípade použitia UVM metodológie je to funkcia *set*)
- Argument volanej funkcie nesmie byť objekt (s výnimkou konštruktu *null*)

3.2 Inicializácia

Emulátory triedy Veloce Strato podporujú inicializáciu signálov pri deklarácii alebo priradením v bloku *initial*. Takýto blok môže byť implementovaný ako implicitné FSM, teda s čakaním na hodinový signál medzi priradeniami. [22]

3.3 Transakcie v emulácii

Štandardne sa pre transakcie v UVM metóde využívajú dáta zabalené v triede. Má to zrejmé výhody – spolu s dátami je možné definovať metódy na prácu s nimi. Príkladom je metóda *convert2string*, ktorá je definovaná ako virtuálny prototyp pre každú triedu odvodenú od triedy *uvm_object* [25]. Jej návratová hodnota je dátový typ *string* a po

implementovaní slúži na konverziu dát v transakčnom objekte na textový reťazec, ktorý je potom možné vypísať do textového záznamu o priebehu simulácie či emulácie. Taktiež je možné definovať ľubovoľné ďalšie metódy. Počas implementácie UVP sa ako užitočné ukázali metódy na nastavenie, alebo naopak, získanie dát z transakčného objektu. Takéto metódy, v kombinácii s obmedzením externého prístupu k dátam, predstavujú vhodnejšiu alternatívu oproti priamemu prístupu k dátam objektu.

Avšak triedy nie sú syntetizovateľné. Emulátory triedy Veloce Strato ich ako parametre v metódach slúžiacich na komunikáciu medzi doménami nepodporujú. Syntéza vyžaduje, aby tieto parametre boli jedným z *packed* dátových typov.

Dátové typy z kategórie *packed* rozdeľujú vektor do podpolí, ku ktorým je možné pristupovať ako k prvkom poľa a dôsledkom toho je, že takýto vektor bude v pamäti uložený ako súvislá množina bitov. Okrem užívateľom definovaných vektorov, ktoré môžu byť aj viacrozmerné, patria do kategórie *packed* dátové typy *byte, shortint, int, longint, integer* a *time*. Taktiež sa môžu skladať z dátových typov *enumerated* alebo iných *packed* štruktúr a vektorov. [30]

Aby bolo možné prenášať transakčné dáta kompaktne v jednom argumente metódy, je vhodné použiť *packed* štruktúru. Táto štruktúra bude predstavovať dáta v transakčnom objekte. Konverzia transakčného objektu na *packed* štruktúru, alebo naopak, prebieha tesne pred odoslaním, alebo tesne po prijatí transakcie v HVL doméne, volaním relevantných metód v transakčnom objekte (pozri prílohu *B*). Vo výsledku teda HVL doména pracuje s objektmi a HDL doména so štruktúrami, pričom oboje obsahujú tie isté dáta. Vzhľadom na to, že HDL doména tieto dáta iba konvertuje z vyššej úrovne abstrakcie na signálové sekvencie (protokol), a naopak, použitie štruktúry namiesto objektu sa javí ako postačujúce.

Vhodným miestom na definovanie obsahu tejto štruktúry je knižnica pre daného *agenta*. Dôvodom je, že je v ňom zahrnutá trieda pre *monitor, driver* a taktiež trieda s transakčnými dátami – všetky tieto triedy budú definovanú štruktúru využívať. Príkladom takejto implementácie sú vybrané časti kódu v prílohe *B*.

3.3.1 Alternatívne transakcie

Aj keď použitie sekvenceru a sekvencií je štandardným prístupom pri implementácii transaktorov v UVM metodológii, nie je to nevyhnutné (metóda *run_phase proxy* triedy pre daný *driver* bude prázdna). Takisto nie je nutné implementovať spúšťanie monitorovacích *forever* cyklov v BFM modeloch (metóda *run_phase proxy* triedy pre daný *monitor* bude prázdna).

Príkladom v priloženom UVP je *agent* modelu oscilátora. Ovládanie BFM modelu prebieha výhradne volaním metód v konfiguračnom objekte tohto *agenta* (*mtrd_osc_agent_config*). Takáto implementácia je možná, pretože *driver* BFM model generuje hodinový signál autonómne a volané metódy slúžia len na úpravu striedy.

Ďalší prípad, kedy môže byť výhodné odkloniť sa od štandardného prístupu rovnakým spôsobom je *agent* modelu vstupných a výstupných portov DUT. Aj v tomto prípade môže byť dostatočné vyčítať alebo zapísať hodnoty vstupov a výstupov použitím metód v konfiguračnom objekte (*mtrd_disc_io_agent_config*). Pokiaľ by však bolo potrebné implementovať určité sekvencie riadenia hodnôt na vstupoch DUT, alebo vyčítať hodnoty na výstupoch, prípadne vstupoch, v určitom poradí, je prehľadnejšie použiť sekvencie a sekvencer, namiesto niekoľkonásobného volania metód v konfiguračnom objekte.

3.3.2 Vplyv počtu transakcii na dobu behu

Nevýhody rozdelenia UVP na dve výpočtové zariadenia v prípade emulácie sa objavia pri veľkom počte transakcii medzi nimi. V tabuľke 3.1 je porovnanie rovnakého testu s dvomi rôznymi verziami BFM modelu transaktora *monitor* (*mtrd_sar_monitor_bfm*) pre prevodník s postupnou aproximáciou (SAR). Pre porovnanie bol použitý test kontrolného obvodu *watchdog* (*mtrd_wdg_test*), ktorý pre vykonanie všetkých stimulov potrebuje simulovať (alebo emulovať) 12,5 sekundy chodu DUT. Prevodník SAR implementovaný v DUT vykoná jeden prevod každých 700 nanosekúnd. To vysvetľuje obrovský počet transakcii vo verzii BFM modelu, ktorá posiela transakciu s aktuálnym výsledkom prevodu do HVL domény po každom jednom prevode. V optimalizovanej verzii bola upravená logika BFM modelu tak, aby posielal transakciu do HVL domény len pokiaľ sa výsledok prevodu pre daný kanál zmenil. Keďže test *mtrd_wdg_test* nie je zameraný na verifikáciu prevodníka SAR, ale na verifikáciu kontrolného obvodu *watchdog*, počet transakcii z BFM modelu sa znížil na 9 (4 po každom z dvoch resetov DUT a 1 pri zmene vstupu kanála, ktorý meria hodnotu napájacieho napätia, čo test využíva na vstup do stavu *FAIL* hlavného FSM). Celkové počty transakcii v tabuľke 3.1 vyjadrujú počet volaní metódy *notify_transaction* v *proxy* triedach každého BFM modelu v UVP. Metódy *notify_transaction* v HVL doméne slúžia na spracovanie prichádzajúcich transakcii z HDL domény.

Tabuľka 3.1 Vplyv počtu transakcii na rýchlosť emulácie

Zariadenie	**Celkový počet transakcii z HDL do HVL**	**Počet transakcii z BFM modelu *monitor* pre SAR**	**Doba behu [s]**			
			HW[1]	**SW[2]**	**KOM.[3]**	**Celkovo**
Simulátor	85	9	-	-	-	418
Emulátor			160	4	56	220
Simulátor	17 858 083	17 858 007	-	-	-	1056
Emulátor			160	111	880	1151

1 – Čas po inicializácii emulátoru strávený vykonávaním v emulátore

2 – Čas po inicializácii emulátoru strávený vykonávaním vo výpočtovej jednotke (*co-model*)

3 – Čas po inicializácii emulátoru strávený komunikovaním medzi doménami

Počet transakcii v optimalizovanej verzii sa znížil o 9 rádov. Z tabuľky 3.1 je zrejmé, že obmedzenie počtu transakcii medzi doménami je kľúčové k odomknutiu potenciálu emulátora. Zatiaľ čo v prípade bez optimalizácii bol emulátor pomalší (!) o 9 % ako simulátor, v emulácii s optimalizovaným počtom transakcii trval identický test iba 52,6 % doby behu simulácie. Emulátor dosiahol vďaka optimalizácii o 80,9 % kratšiu dobu behu oproti neoptimalizovanej verzii, zatiaľ čo pre simulátor bolo toto zrýchlenie iba 60,4%.

Okrem toho, že obrovský počet transakcii predĺžil dobu komunikácie medzi zariadeniami, bol mnohonásobne zvýšený aj potrebný čas vo výpočtovej jednotke na ich spracovanie (stĺpec SW v tabuľke 3.1).

Dáta v tabuľke 3.1 o počte transakcii a dobe behu emulátora boli získané zo súboru so záznamom o priebehu emulácie s názvom *simulation_profile.log*.

3.4 Posuv simulačného času

Ako už bolo spomenuté vyššie, posuv simulačného času je v prípade UVP povolený len synchronizáciou na hodinový signál v HDL doméne. Na takúto synchronizáciu stačí v jednom z BFM modelov, ktoré majú prístup k hodinovému signálu, implementovať veľmi jednoduchú metódu. Pre tieto účely je vhodnejší *monitor* (*mtrd_osc_monitor_bfm*), pretože *driver* nie je prítomný v pasívnom režime *agenta*.

```
1: task wait_clk (int unsigned num);
2: // pragma tbx xtf
3:   repeat (num) @(posedge clk_bfm);
4: endtask
```

Zdrojový kód 3.9 Metóda na posuv simulačného času v HDL doméne

V HVL doméne je potrebné implementovať metódu v niektorej z tried, ktoré majú prístup k ukazovateľu na daný BFM model. V priloženom UVP je táto metóda implementovaná v konfiguračnej triede pre *agenta* modelu oscilátora (*mtrd_osc_agent_config*). Táto trieda bola zvolená, pretože sú v nej uložené ukazovatele na BFM modely pre trasanktory *monitor* (*mon_bfm*) a taktiež *driver* (*drv_bfm*).

```
1: task wait_clk (int unsigned num);
2:   mon_bfm.wait_clk(num);
3: endtask
```

Zdrojový kód 3.10 Metóda na posuv simulačného času v HVL doméne

Prístup k tejto metóde z virtuálnej sekvencie je zabezpečený predaním ukazovateľa na danú konfiguračnú triedu. Pri predávaní ukazovateľov naprieč UVP postaveným na metodológii UVM je vhodné dodržať princípy hierarchie a znovupoužiteľnosti. Takého predávanie ukazovateľov prebieha vždy v triede nadradenej obom triedam (tej, do ktorej chceme daný ukazovateľ uložiť a tej, na ktorú daný ukazovateľ ukazuje). Z toho dôvodu bolo v priloženom UVP toto predanie implementované v základnej triede, z ktorej sú odvodené všetky testy (*mtrd_test_base*), v metóde s názov *init_vseq*. V prípade, že je

hlavná virtuálna sekvencia zložená z ďalších sekvencii, či už virtuálnych alebo nie, aby bol dodržaný princíp hierarchie a znovupoužiteľnosti, propagácia ukazovateľov sekvenciám nižšie v hierarchii prebieha vždy z nadradenej sekvencie o jednu úroveň vyššie. Každá sekvencia s ukazovateľom na konfiguračnú triedu môže volať všetky metódy v nej implementované. Príklad volania synchronizačnej metódy zo sekvencie je uvedený na nasledujúcej ukážke zdrojového kódu.

```
1: m_osc_cfg.wait_clk(10);
```

Zdrojový kód 3.11 Volanie metódy na posuv simulačného času zo sekvencie

Vzhľadom na to, že metóda implementovaná v BFM modeli je syntetizovaná, na počítanie ubehnutých periód vznikne v programovateľnej logike emulátoru čítač. Veľkosť tohto čítača je možné ovplyvniť dátovým typom, ktorý vstupuje do metódy *repeat* – teda dátovým typom vstupu metódy *wait_clk* (rovnaký pre metódu v HVL aj HDL doméne).

Využitie jednotlivých prvkov je po syntéze dostupné v súbore *area.report* vygenerovanom do adresáru *veloce.med/rtlc.out*. Počíta sa využitie vyhľadávacích tabuliek – Lookup Table (LUT) a klopných obvodov – Flip-Flop (FF). Priložený BFM model pre *monitor* oscilátora obsahoval v čase písania tejto kapitoly len metódu *wait_clk* a ukazovateľa na *proxy* triedu. Porovnanie počtu využitých prvkov LUT a FF pre rôzne dátové typy je uvedené v tabuľke 3.2.

Tabuľka 3.2 Využité prvky LUT a FF pre rôzne veľké čitače

Dátový typ (*unsigned*)	**Veľkosť v bitoch**	**Počet využitých prvkov LUT celkovo pre BFM model *mtrd_osc_monitor_bfm***	**Počet využitých prvkov FF celkovo pre BFM model *mtrd_osc_monitor_bfm***
shortint	16	172	56
int	32	256	88
longint	64	432	152

Tabuľka 3.2 potvrdzuje očakávania, že čím viac bitov bude čítač mať, tým viac prvkov využije. Bolo by preto logické použiť dátový typ s čo najmenším počtom bitov. Avšak v porovnaní s celkovým počtom využitých prvkov modelmi, DUT a okolitou logikou, ktorá je nutná pre emuláciu a je generovaná automaticky, sú tieto rozdiely zanedbateľné. V prípade použitia čítača s veľkosťou 32 bitov bol celkový počet využitých prvkov LUT 52 171 a prvkov FF 33 992. To, že sú tieto rozdiely zanedbateľné potvrdzuje aj doba behu simulácie a emulácie, ktorá sa pri zmene 16 bitového čítača na 64 bitový nepredĺžila o viac ako 2 %, čo nemuselo byť nevyhnutne spôsobené zväčšením čítača, ale inými náhodnými faktormi. Aj napriek tomu sa 32 bitový čítač javí ako plne dostačujúci, keďže pri perióde hodinového signálu 50 nanosekúnd dokáže na jedno zavolanie zabezpečiť posun o viac ako 214 sekúnd.

3.4.1 Paralelná synchronizácia na hodinový signál

Pokiaľ je v teste nutné využiť niekoľko paralelných posuvov času (použitím konštruktu *fork-join*), vo verifikačnom prostredí určenom výhradne pre simuláciu je možné označiť metódy synchronizujúce sa na hodinový signál (*wait_clk* v predchádzajúcom texte) kľúčovým slovom *automatic*. Tým sa zabezpečí, že každé volanie metódy si vytvorí vlastnú inštanciu premenných potrebných na počítanie periód hodinového signálu. V prípade emulácie však nie je možné upraviť počet syntetizovaných čítačov za behu. Prípadný pokus o syntézu takejto metódy skončí chybou TBXC-16497. Riešením je implementácia viacerých paralelných metód, ktoré je potom možné nezávisle volať z HVL domény. V takomto prípade by už rozdiel medzi počtom využitých prvkov LUT a FF nemusel byť zanedbateľný a je potrebné zvážiť, či je daný test vhodné emulovať.

Nutnosť implementovať paralelne viacero metód pri využití emulácie platí iba pre HDL doménu. V HVL doméne je aj pri emulácii možné, aby bola metóda označená kľúčovým slovom *automatic* a jeden z jej parametrov rozhodoval o tom, ktorý čítač v HDL doméne použiť.

```
1: task automatic wait_clk (int unsigned num, int unsigned counter);
2:   case (counter)
3:     1: mon_bfm.wait_clk_1(num);
4:     2: mon_bfm.wait_clk_2(num);
5:   endcase
6: endtask
```

Zdrojový kód 3.12 Metóda na paralelný posuv simulačného času v HVL doméne

3.5 Asociatívne polia

Pri písaní knižníc pre UVP je potrebné počítať s tým, že knižnica obsahujúca asociatívne polia nemôže byť importovaná v HDL doméne (príkaz *velanalyze* skončí chybou TBXC-6375). Toto obmedzenie je možné riešiť dvoma spôsobmi: 1) skompilovať takéto knižnice iba pre HVL doménu; 2) označiť asociatívne polia ako *flexmem*

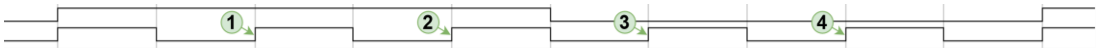
Flexmem slúži na stránkovanie veľkých pamätí, pričom v HVL doméne je uchovaná celá pamäť a do HDL domény sa presúva vždy iba jej relevantná časť pre daný moment [29]. Používanie *flexmem* nie je relevantné pre túto prácu – odporúča sa používať pre pamäte od veľkosti 16 MB [29]. Aj keď takéto označenie asociatívneho poľa umožní knižnicu importovať v HDL doméne, jeho použitie v tejto doméne bude obmedzené – nie je možné pristupovať k asociatívnym poliam obsahujúcim dáta typu *string* alebo *real*. Syntéza by v takom prípade skončila chybou TBXC-6758. Taktiež dátový typ indexu je obmedzený na *bit* alebo bitový vektor.

Vzhľadom na to, že asociatívne polia nachádzajú svoje uplatnenie prakticky výhradne v HVL doméne a ich importovanie do HDL domény predstavuje zásadné obmedzenie možností, vytvorenie samostatnej knižnice so všetkými potrebnými asociatívnymi poľami sa v tomto prípade javí ako lepšia voľba.

3.6 Asynchrónny stimul

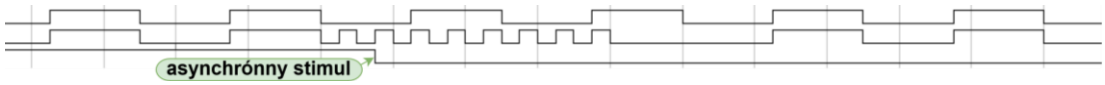
Niektorá funkcionalita návrhu vyžaduje pre svoju verifikáciu asynchrónny stimul. Typicky to môže byť asynchrónny reset, resynchronizácia asynchrónnych vstupov alebo potlačenie zákmitov (*debouncer*). Bežným riešením v simulácii je použitie posuvu simulačného času konštruktom #. Ako už bolo viackrát spomenuté, v emulácii táto možnosť nie je k dispozícii.

Možným riešením je použitie hodinového signálu s rozdielnou periódou ako je perióda hodinového signálu pre DUT. Napríklad pri generovaní hodinového signálu s päťnásobnou frekvenciou tak vzniknú štyri časové body, asynchrónne voči hodinovému signálu pre DUT, na ktoré je možné synchronizovať stimul. Takýto stimul bude asynchrónny voči hodinovému signálu pre DUT.



Obrázok 3.1 Hodinový signál s päťnásobnou periódou

Toto riešenie má však za následok päťnásobný nárast počtu bodov, ktoré musí emulátor spracovať, čo sa prejaví spomalením emulácie. Je teda žiadúce obmedziť generovanie rýchleho hodinového signálu len na časové úseky kedy je potrebné generovať asynchrónny stimul. V ostatných časových úsekoch je tento hodinový signál možné „skryť“ za hodinový signál generovaný pre DUT – teda zosynchronizovať ich tak, aby ich nábežné hrany prichádzali v rovnakom čase. Implementácia tejto metódy bola použitá v priloženom UVP v teste pre overenie funkcionality potlačenia zákmitov (test *mtrd_debouncer_test*).

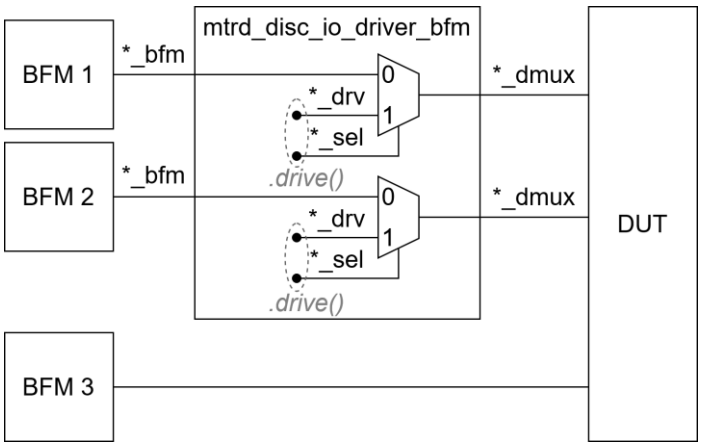


Obrázok 3.2 Selektívne generovanie rýchleho hodinového signálu

3.7 Neštandardné riadenie vstupov návrhu

Pri verifikácii návrhu môže nastať situácia, kedy je potrebné riadiť vstupy DUT iným spôsobom, ako je definované správanie v BFM modeloch. Pre riešenie tohto problému existuje viacero prístupov. Napríklad jednotlivé BFM modely môžu implementovať metódy pre explicitné nastavenie výstupov. V implementácii priloženého UVP bolo zvolené centralizované riešenie, kde relevantné signály prechádzajú cez jeden z BFM modelov (*mtrd_disc_io_driver_bfm*) a riadením multiplexorov je možné zvoliť zdroj každého signálu jednotlivo. Volaním metódy *drive* je možné voliť medzi bežným výstupom BFM modelu alebo nastavenou hodnotou. Principiálna schéma implementácie je uvedená na obrázku 3.3. Takéto riadenie vstupov DUT bolo v priloženom UVP využité v teste pre overenie funkcionality potlačenia zákmitov (test *mtrd_debouncer_test*).

V prípade potreby je takéto riešenie možné rozšíriť o sofistikovanejšie riadenie (napríklad pre účely testability) alebo viacvstupové multiplexory.



Obrázok 3.3 Multiplexovanie vstupov DUT

3.8 Hierarchický prístup k vnútorným signálom návrhu

Vďaka 100% viditeľnosti v emulátoroch triedy Veloce Strato je podobne ako v simulácii možné hierarchicky pristupovať k ľubovoľným signálom vnútri DUT. Jediným rozdielom je, že tento prístup nie je možné vykonať z HVL domény. Je preto nutné pre takýto prístup vytvoriť *agenta*, ktorého BFM model bude v HDL doméne (v priloženom UVP je to *mtrd_int_mon_bfm*). BFM model pri volaní relevantnej metódy vráti hodnotu z vopred definovanej hierarchickej cesty.

3.9 Emulácia návrhu na hradlovej úrovni

Emulácia sa vykonáva v hardvéri a preto musí vždy najskôr prebehnúť syntéza návrhu. Oproti simulácii na RTL úrovni sa to prejaví ako nevýhoda v tom, že syntéza trvá dlhšie ako kompilácia pre simuláciu. Zatiaľ čo kompilácia priloženého referenčného návrhu na RTL úrovni trvá rádovo sekundy, v prípade syntézy pre emulátor sa jedná rádovo o minúty. V porovnaní so simuláciou na hradlovej úrovni má však emulátor výhodu v tom, že takýto typ emulácie sa bude vykonávať rovnako rýchlo ako v prípade emulácie syntetizovaného návrhu na RTL úrovni. To pre simuláciu neplatí, ako dokazuje tabuľka 3.3. Je však potrebné počítať s tým, že po syntéze návrhu externým programom je pre emuláciu nutné syntézu vykonať znova. Vstupom syntézy pre emuláciu je knižnica hradiel a popis návrhu na hradlovej úrovni, ktorý je v prípade simulácie dostačujúci. Tým pádom sa ani v tomto prípade nie je možné vyhnúť dlhšej príprave vstupov pre emuláciu.

Tabuľka 3.3 Porovnanie rýchlosti simulácie a emulácie na rôznych úrovniach

Úroveň	**Zariadenie**	**Doba behu[1] [s]**
RTL	Simulátor	499
	Emulátor	220
Hradlová	Simulátor	3952
	Emulátor	220

1 – V prípade emulácie je uvedený čas po inicializácii emulátoru

Zatiaľ čo sa čas potrebný pre simuláciu zvýšil takmer osemkrát, čas potrebný pre emuláciu ostal identický. Vo výsledku emulácia na hradlovej úrovni dosiahla takmer 18-násobné zrýchlenie oproti simulácii. Pre porovnanie bol opäť použitý test *mtrd_wdg_test*.

3.10 Emulácia malých návrhov

Najmenšou nedeliteľnou hardvérovou jednotkou, ktorá môže byť užívateľovi priradená je 1 AVB s kapacitou 40 miliónov ASIC ekvivalentných hradiel (pozri kapitolu *1.4.3 Siemens Veloce Strato*). Vzhľadom na to, že veľkosť návrhov, ktorým sa venuje táto práca, je do 100 tisíc ASIC ekvivalentných hradiel, je kapacita 1 AVB zbytočne veľká. Takmer celá jej kapacita je pri emulácii nevyužitá. Z ekonomického, ale aj časového hľadiska by bolo výhodnejšie využiť kapacitu 1 AVB z väčšej časti. V ideálnom prípade by sa do kapacity 1 AVB zmestilo 400 takýchto návrhov paralelne. V skutočnosti bude toto číslo menšie, pretože hardvérové zdroje AVB sú obmedzené a vo výpočte nebola započítaná kapacita, ktorú zaberie okolitá logika nutná pre chod emulátoru. Úzkym hrdlom paralelnej emulácie bude znásobený počet transakcií medzi doménami. V kombinácii s faktom, že generovanie užívateľských hodinových signálov je počas komunikácie zastavené, to môže mať za následok výrazné spomalenie emulácie jednotlivých testov. Taktiež sa predĺži aj čas potrebný na syntézu.

Takáto paralelizácia nie je súčasťou UVM metodológie a jej implementácia tým pádom nemusí byť triviálna. To však neznamená že je to nemožné. Implementácia prostredia pre paralelnú emuláciu je jedna z oblastí, ktoré môžu byť záujmom pokračovania tejto práce.

ZÁVER

Pre hardvérovú akceleráciu bol použitý emulátor Veloce Strato TiL v20.0.2, pretože pre účely verifikovania návrhov digitálnych častí integrovaných obvodov so zmiešanými signálmi je výhodnejšia 100% viditeľnosť a ovládateľnosť vnútorných signálov návrhu, aj za cenu nižšej rýchlosti oproti FPGA prototypovaniu.

Aby bolo možné overiť adekvátnosť UVP postaveného na UVM metodológii, bolo potrebné implementovať referenčný návrh s parametrami bežnými pre návrhy automobilového priemyslu. Tieto parametre sú uvedené v druhej kapitole a v realizovanom referenčnom návrhu boli splnené. Pri implementácii UVP pre referenčný návrh sa UVM metodológia osvedčila ako vhodná pre využitie na tento účel.

Ďalej boli preskúmané rôzne vplyvy na rýchlosť simulácie a emulácie. V kapitole *3.1.1* je vysvetlená jedna z najdôležitejších optimalizácii, na ktorú je potrebné pri implementovaní návrhu a UVP myslieť, pretože pri správnom využití umožňuje zdvojnásobiť rýchlosť emulácie. Z výsledkov v tabuľke *3.1* je zrejmé, že rýchlosť emulácie je viac postihnutá pri vysokom počte transakcií ako rýchlosť simulácie za rovnakých podmienok.

Práca dokazuje, že je možné vysporiadať sa s mnohými problémami, ktoré súvisia s využitím emulácie. V určitých prípadoch však riešenie nemusí byť ideálne. Testy s asynchrónnym stimulom sú v emulácii realizovateľné, ale ich popis je komplikovanejší ako v prípade simulácie. Simulácia sa javí ako lepšia voľba napríklad pokiaľ test potrebuje vykonať väčšie množstvo paralelných posuvov času alebo pracuje vo veľkej miere s asynchrónnym stimulom. Najmä pokiaľ ide o testy s krátkym simulovaným časom, ktorých rýchlosť je aj v simulácii dostačujúca. Problémy môžu nastať s emuláciou verifikačných IP jadier, ktorých syntéza môže byť nerealizovateľná. V takom prípade je potrebné použiť simuláciu.

Najväčšie zrýchlenie bolo zaznamenané v prípade emulácie popisu návrhu na hradlovej úrovni, pričom emulátor dosiahol 18-násobnú rýchlosť oproti simulácii. Rýchlosť emulácie popisu na hradlovej úrovni je takmer identická s rýchlosťou emulácie popisu na RTL úrovni. Emulácia popisu návrhu na RTL úrovni dosiahla zhruba dvojnásobné zrýchlenie pre implementované testy.

Zo skúseností s emulátormi triedy Veloce Strato je zrejmé, že najlepšie využitie nájdu pri dlhých a synchrónnych testoch, ako môže byť napríklad test kontrolného obvodu *watchdog* alebo test prevodníku SAR. V prípade krátkych testov je pravdepodobné, že syntéza návrhu pre emuláciu a nahranie dát do emulátora zaberie viac času ako kompilácia a simulácia.

Vo výsledku je teda vhodné rozdeliť testy do dvoch skupín a paralelne využiť simuláciu a emuláciu pre dosiahnutie čo najrýchlejšej verifikácie.

LITERATÚRA

[1] 2020 Wilson Research Group functional verification study: IC/ASIC functional verification trend report. Siemens Digital Industries Software [online]. 2020 [cit. 2022-01-02]. Dostupné z: https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study-ic-asic-fucntional-verification-trend-report

[2] LAM, William. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall PTR, 2005. ISBN 0-13-143347-4.

[3] WANG, Laung-Terng, Yao-Wen CHANG a Kwang-Ting CHENG. Electronic Design Automation: Synthesis, Verification, and Test. Morgan Kaufmann, 2009. ISBN 978-0123743640.

[4] RIZZATTI, Lauro. Hardware Emulation: A Weapon of Mass Verification. Electronic Design [online]. 29. Október 2014 [cit. 2021-12-30]. Dostupné z: https://www.electronicdesign.com/technologies/test-measurement/article/21800385/hardware-emulation-a-weapon-of-mass-verification

[5] Getting Started with Formal-Based Technology. *Verification Academy* [online]. [cit. 2021-11-21]. Dostupné z: https://verificationacademy.com/courses/Getting-Started-with-Formal-Based-Technology

[6] Formal Assertion-Based Verification. *Verification Academy* [online]. [cit. 2021-11-01]. Dostupné z: https://verificationacademy.com/courses/Formal-Assertion-Based-Verification

[7] Questa Formal Verification Apps. *Siemens Digital Industries Software* [online]. [cit. 2021-10-30]. Dostupné z: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/

[8] Coverage. *Verification Academy* [online]. 2019 [cit. 2021-11-20]. Dostupné z: https://verificationacademy.com/cookbook/coverage

[9] VERMA, Gaurav Kumar a Doug WARMKE. *Supercharge Your Verification Using Rapid Expression Coverage as the Basis of a MC/DC-Compliant Coverage Methodology* [online]. DVCon 2014, 2014 [cit. 2021-11-22]. Dostupné z: https://verificationacademy.com/resources/technical-papers/supercharge-your-verification-using-rapid-expression-coverage-basis-mc/dc-compliant-coverage-methodology

[10] BRUNET, Jean-Marie. Veloce Hardware-Assisted Verification – Complete, Unified, and Progressive. *Verification Horizons* [online]. 2021, September, (zväzok 17, vydanie 2), 37-45 [cit. 2021-11-21]. Dostupné z: https://verificationacademy.com/verification-horizons/september-2021-volume-17-issue-2/veloce-hardware-assisted-verification-complete-unified-and-progressive

[11] BRUNET, Jean-Marie a Lauro RIZZATTI. Hardware-assisted Verification Through the Years. *Verification Horizons* [online]. 2021, September, (zväzok 17, vydanie 2), 31-36 [cit. 2021-11-21]. Dostupné z: https://verificationacademy.com/verification-horizons/september-2021-volume-17-issue-2/hardware-assisted-verification-through-the-years

[12] RIZZATTI, Lauro a Gabrielle PULINI. Increase hardware emulation productivity. *Siemens Digital Industries Software* [online]. [cit. 2021-11-28]. Dostupné z: https://resources.sw.siemens.com/en-US/white-paper-increase-hardware-emulation-productivity-with-virtual-mode

[13] SELVIDGE, Charley a Vijay CHOBISA. The Veloce Strato Platform: Unique Core Components Create High-Value Advantages. *Siemens Digital Industries Software* [online]. [cit. 2021-11-22]. Dostupné z: https://resources.sw.siemens.com/en-US/white-paper-the-veloce-strato-platform-unique-core-components-create-high-value

[14] Mentor aims to grow emulation with lower gate-count hardware. *Tech Design Forum* [online]. 5. Apríl 2018 [cit. 2021-11-30]. Dostupné z: https://www.techdesignforums.com/blog/2018/04/05/mentor-strato-emulator-lower-gate-count-modular/

[15] SQUIERS, Ron. Co-modeling – A powerful capability for hardware emulation. *Siemens Digital Industries Software* [online]. [cit. 2021-11-30]. Dostupné z: https://resources.sw.siemens.com/en-US/white-paper-co-modeling-a-powerful-capability-for-hardware-emulation

[16] Logic and circuit simulation. HUANG, Jiun-Lang, Cheng-Kok KOH a Stephen CAULEY. *Electronic Design Automation*. Elsevier, 2009, s. 449-512. ISBN 978-0-12-374364-0.

[17] CHOBISA, Vijay a Sanjay GUPTA. Emulation delivers system-level power verification. *Tech Design Forum* [online]. 26. Október 2012 [cit. 2021-11-27]. Dostupné z: https://www.techdesignforums.com/practice/technique/emulation-system-level-power-verification/

[18] DEMPSEY, Paul. Mentor launches new Strato emulation platform. *Tech Design Forum* [online]. 16. Február 2017 [cit. 2021-12-05]. Dostupné z: https://www.techdesignforums.com/blog/2017/02/16/strato-emulation-mentor-graphics/

[19] VAN DER SCHOOT, Hans a Ahmed YEHIA. UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up. *Verification Academy* [online]. 2015 [cit. 2022-01-02]. Dostupné z: https://verificationacademy.com/news/featured-paper-dvcon-europe-2015

[20] Emulation. *Verification Academy* [online]. 2018 [cit. 2022-01-02]. Dostupné z: https://verificationacademy.com/cookbook/emulation

[21] VAN DER SCHOOT, Hans, Ahmed YEHIA a Michael HORN. *Reducing Design Risk with Testbench Acceleration*. 2016. Whitepaper. Mentor Graphics Corporation.

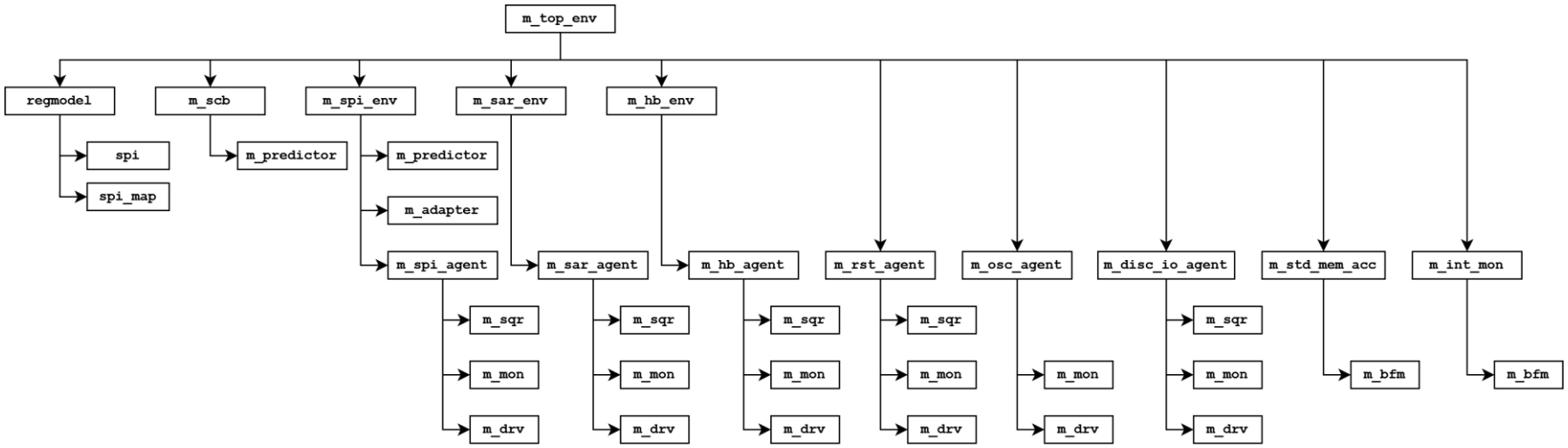
[22] Veloce Languages and Communication User Guide. Vydanie v20.0.2. Siemens, 2020.
[23] VAN DER SCHOOT, Hans, Anoop SAHA, Ankit GARG a Krishnamurthy SURESH. Off to the Races with Your Accelerated SystemVerilog Testbench. *Verification Academy* [online]. 2011 [cit. 2022-01-02]. Dostupné z: https://verificationacademy.com/news/featured-acceleration-techniques-paper-dvcon-2011
[24] Register Assistant UVM User Manual. Vydanie v2019.3. Mentor Graphics Corporation, 2019.
[25] IEEE STD 1800.2™-2017. *IEEE Standard for Universal Verification Methodology: Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, USA, 2017.
[26] Advanced UVM Register Modeling: There's More Than One Way to Skin A Reg. *DVCon 2014* [online]. San Jose, CA, 2014 [cit. 2022-05-16]. Dostupné z: https://www.researchgate.net/publication/275947131_Advanced_UVM_Register_Modeling_-_There%27s_More_Than_One_Way_to_Skin_A_Reg
[27] BORCHERS, Kai, Sergio MONTENEGRO a Frank DANNEMANN. *Volatile Register Handling for FPGA Verification Based on SVAs Incorporated into UVM Environments* [online]. 2019 [cit. 2022-05-16]. Dostupné z: https://www.researchgate.net/publication/333919241_Volatile_Register_Handling_for_FPGA_Verification_Based_on_SVAs_Incorporated_into_UVM_Environments
[28] SHIMIZU, Keisuke. Register Access Methods. *ClueLogic* [online]. 2014 [cit. 2022-05-17]. Dostupné z: http://cluelogic.com/2013/02/uvm-tutorial-for-candy-lovers-register-access-methods/
[29] Veloce User Guide. Vydanie v20.0.2. Siemens, 2020.
[30] IEEE STD 1800™-2017. *IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, USA, 2018.

ZOZNAM SKRATIEK

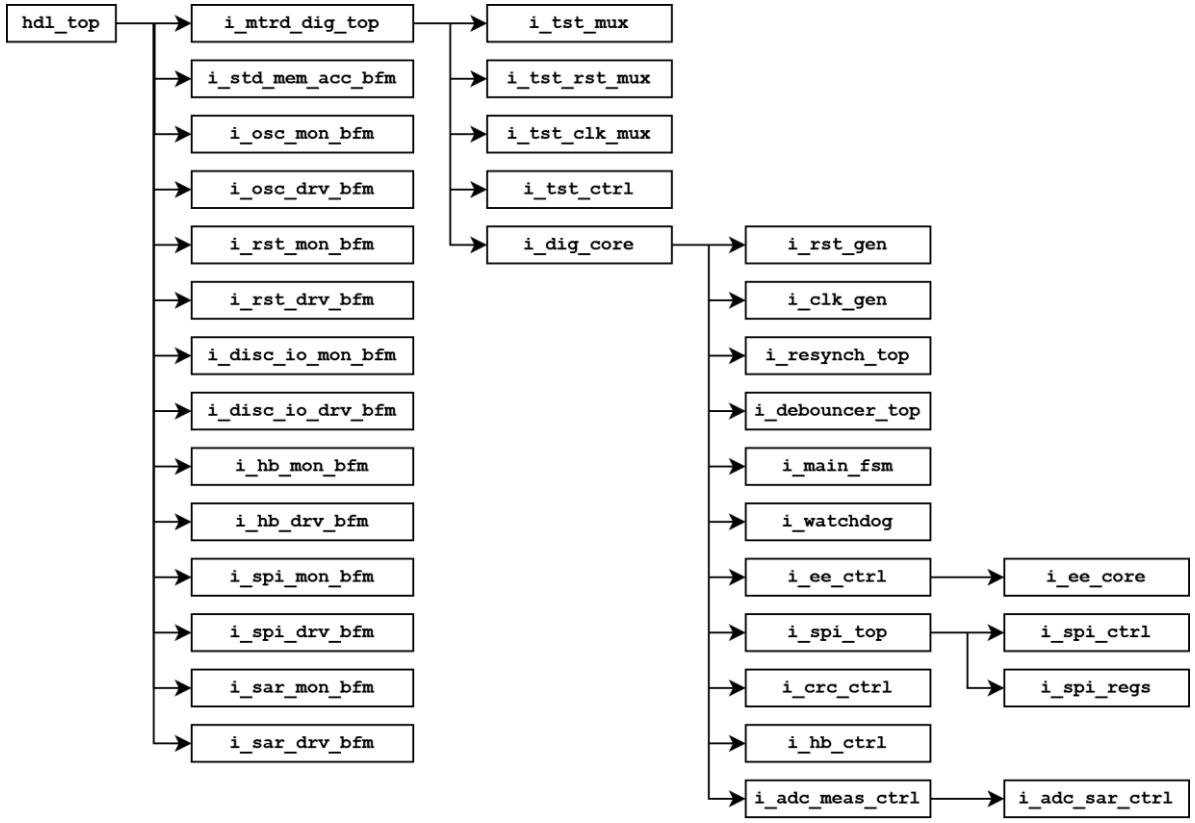
AVB	Advanced Verification Board
BFM	Bus Functional Model
BG	Billion Gates
CEX	Counterexample
CSV	Comma-Separated Values
DUT	Device Under Test
FF	Flip-Flop
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
HDL	Hardware Description Language
HVL	Hardware Verification Language
ICE	In-Circuit Emulation
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
LUT	Lookup Table
MHz	Megahertz
OS	Operačný Systém
RTL	Register-Transfer Level
SAR	Successive Approximation
SLEC	Sequential Logic Equivalence Check
SPI	Serial Peripheral Interface
SVA	SystemVerilog Assertions
UCDB	Unified Coverage Database
UVM	Universal Verification Methodology
UVP	Unifikované Verifikačné Prostredie
W1C	Write-1-to-Clear

ZOZNAM PRÍLOH

Príloha A Hierarchia implementácie



Obrázok A.1 Štruktúra implementovaného referenčného návrhu



Obrázok A.2 Štruktúra implementovaného referenčného návrhu

Príloha B Vybrané časti zdrojového kódu

```
1:  package mtrd_spi_agent_pkg;
2:    ...
3:    typedef struct packed {
4:      int bits;
5:      t_spi_buffer req_buffer;
6:      t_spi_buffer rsp_buffer;
7:    } mtrd_spi_seq_item_s;
8:
9:    `include "mtrd_spi_seq_item.sv"
10:   `include "mtrd_spi_agent_config.sv"
11:   `include "mtrd_spi_driver.sv"
12:   `include "mtrd_spi_monitor.sv"
13:   typedef uvm_sequencer #(mtrd_spi_seq_item) mtrd_spi_sequencer;
14:   `include "mtrd_spi_agent.sv"
15:   `include "mtrd_spi_seq.sv"
16:
17: endpackage : mtrd_spi_agent_pkg
```

Zdrojový kód B.1 Úryvok mtrd_spi_agent_pkg

```
1:  class mtrd_spi_seq_item extends uvm_sequence_item;
2:    ...
3:    protected mtrd_spi_seq_item_s   data;
7:    ...
8:  endclass : mtrd_spi_seq_item
9:  ...
10: function void mtrd_spi_seq_item::from_struct (
11:   mtrd_spi_seq_item_s s
12: );
13:   data = s;
14: endfunction
15: ...
16: function mtrd_spi_seq_item_s mtrd_spi_seq_item::to_struct();
17:   return data;
18: endfunction
```

Zdrojový kód B.2 Úryvok mtrd_spi_seq_item

```
1:  class mtrd_spi_driver extends uvm_driver #(mtrd_spi_seq_item);
2:    ...
3:    local virtual mtrd_spi_driver_bfm m_bfm;
4:    ...
5:    uvm_analysis_imp #(mtrd_spi_seq_item, mtrd_spi_driver) ai;
6:    mtrd_spi_seq_item item_rsp;
10:   ...
11:  endclass : mtrd_spi_driver
12:  ...
13:  task mtrd_spi_driver::run_phase (uvm_phase phase);
14:    mtrd_spi_seq_item item;
15:    mtrd_spi_seq_item_s item_s;
16:
17:   forever begin
18:     seq_item_port.get_next_item(item);
19:     `uvm_info("SPI_DRV_BFM", item.convert2string(), UVM_HIGH)
20:     item_s = item.to_struct();
21:     m_bfm.drive(item_s);
22:     item.copy_rsp_buffer(item_rsp);
23:     seq_item_port.item_done();
24:   end
25: endtask
26:
27: function void mtrd_spi_driver::write (mtrd_spi_seq_item item);
28:   item_rsp = item;
29: endfunction
```

Zdrojový kód B.3 Úryvok mtrd_spi_driver

```
1:  import mtrd_spi_agent_pkg::mtrd_spi_seq_item_s;
2:
3:  interface mtrd_spi_driver_bfm (
4:    input   logic   clk_bfm,
5:    output  logic   mosi,
6:    output  logic   sclk,
7:    output  logic   csb
8:  );
9:  // pragma attribute mtrd_spi_driver_bfm partition_interface_xif
10:   ...
11:   task drive (mtrd_spi_seq_item_s item_s);
12:   // pragma tbx xtf
13:   ...
14:   endtask : drive
15:   ...
16: endinterface : mtrd_spi_driver_bfm
```

Zdrojový kód B.4 Úryvok mtrd_spi_driver_bfm

```
1:  class mtrd_spi_monitor extends uvm_monitor;
2:    ...
3:    local virtual mtrd_spi_monitor_bfm m_bfm;
4:    ...
5:    uvm_analysis_port #(mtrd_spi_seq_item) ap;
6:    mmtrd_spi_seq_item item
7:    ...
8:  endclass : mtrd_spi_monitor
9:  ...
10: task mtrd_spi_monitor::run_phase (uvm_phase phase);
11:   m_bfm.run();
12: endtask
13: ...
14: function void mtrd_spi_monitor::notify_transaction (
15:   mtrd_spi_seq_item_s item_s
16: );
17:   item.from_struct(item_s);
18:   `uvm_info("SPI_MON_BFM", item.convert2string(), UVM_HIGH)
19:   ap.write(item);
20:   item = mtrd_spi_seq_item::type_id::create("item");
21: endfunction
```

Zdrojový kód B.5 Úryvok mtrd_spi_monitor

```
1:  import mtrd_spi_agent_pkg::mtrd_spi_seq_item_s;
2:  import mtrd_spi_agent_pkg::mtrd_spi_monitor;
3:
4:  interface mtrd_spi_monitor_bfm (
5:    input   logic   clk_bfm,
6:    input   logic   mosi,
7:    input   logic   miso,
8:    input   logic   miso_hiz,
9:    input   logic   sclk,
10:   input   logic   csb
11: );
12: // pragma attribute mtrd_spi_monitor_bfm partition_interface_xif
13:
14:   mtrd_spi_monitor    proxy;
15:   ...
16:   task run ();
17:   // pragma tbx xtf
18:     ...
19:   endtask : run
20:   ...
21: endinterface : mtrd_spi_monitor_bfm
```

Zdrojový kód B.6 Úryvok mtrd_spi_monitor_bfm

Príloha C CD so zdrojovým kódom

Obsah CD:

src.zip - Zdrojový kód unifikovaného verifikačného prostredia a referenčného návrhu