VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VERIFIKACE NÁSTROJŮ PRO PROTOKOL FRAMELINK V SYSTEMVERILOGU

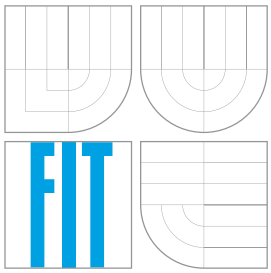
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK SANTA

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VERIFIKACE NÁSTROJŮ PRO PROTOKOL FRAMELINK V SYSTEMVERILOGU

SYSTEMVERILOG VERIFICATION OF FRAMELINK PROTOCOL TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK SANTA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VIKTOR PUŠ

BRNO 2009

Abstrakt

Vyhnout se chybám při vývoji číslicových systémů je téměř nemožné. Přitom brzké odhalení takových chyb pomáha šetřit čas i peníze. Tato práce se zabývá funkční verifikací různých nástrojů na spracování dat. Nejdřív jsou diskutovány principy a postupy funkční verifikace, následně je vytvořen návrh a implementace verifikačního prostředí v jazyce SystemVerilog. Na závěr jsou shrnuty výsledky verifikace.

Abstract

In the development process of digital circuits, it is often not possible to avoid introducing errors into systems that are being developed. Early detection of such errors saves money and time. This thesis deals with functional verification of various data processing components. General functional verification principles and practices are discussed and design and implementation of a SystemVerilog verification environment is described in detail. The verification results are summarized and evaluated.

Klíčová slova

verifikace, SystemVerilog, FrameLink

Keywords

verification, SystemVerilog, FrameLink

Citace

Marek Santa: Verifikace nástrojů pro protokol FrameLink v SystemVerilogu, bakalářská práce, Brno, FIT VUT v Brně, 2009

Verifikace nástrojů pro protokol FrameLink v SystemVerilogu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Viktora Puše. Další informace mi poskytli moji kolegové z projektu Liberouter. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Santa
18. mája 2009

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce panu Ing. Viktoru Pušovi za odborné vedení a konzultace. Dále bych chtěl poděkovat panu Ing. Petrovi Kobierskému za uvedení do problematiky verifikací v jazyce SystemVerilog a ostatním kolegům z projektu Liberouter za poskytnutí cených informací a rad.

Obsah

Kapitola 1

Úvod

Pod pojmom verifikácia si najčastejšie predstavíme hľadanie chýb. Táto predstava však nie je úplná. Cieľom verifikačného procesu je zistiť, či zariadenie spĺňa úlohu, pre ktoré bolo navrhnuté – teda či dizajn odpovedá špecifikácii. Chyby sú práve to, čo dostávame ako výsledok nezrovnalostí medzi špecifikáciou a výsledným produktom.

Podľa [6] môžeme chyby vo všeobecnosti rozdeliť do dvoch hlavných kategórií. Prvú tvoria chyby na úrovni bloku (modulu) vytváraného jednou osobou. Sčítava ALU správne dve čísla? Skončí každá zbernicová operácia úspešne? Vytvorenie testov na odhalenie takých chýb je jednoduché, pretože sa nachádzajú v rámci jedného bloku.

Do druhej kategórie spadajú chyby, ktoré sa vyskytujú na hraniciach medzi blokmi. Najčastejšie vznikajú pri spolupráci viacerých dizajnérov, kde každý si vyloží špecifikáciu trocha iným spôsobom. V takom prípade môže verifikácia pomôcť rozhodnúť, ktorý pohľad je správny.

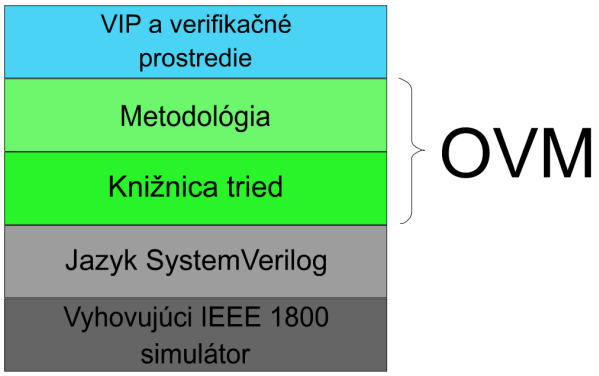
Pri testovaní jediného bloku musíme vytvoriť prostredie, ktoré simuluje všetky okolité komponenty, čím sa stáva do značnej miery komplexným a to prináša ďalšiu možnosť na zanesenie chýb. Verifikáciou zapojenia viacerých blokov, prípadne celého dizajnu, môžme vernejšie napodobiť podmienky, v akých bude dizajn skutočne fungovať a odhaliť tak chyby spojené najmä s paralelným behom viacerých operácií.

Kapitola 2

Verifikácia, SystemVerilog

Pri porovnaní procesu dizajnu a verifikácie zistíme, že časovo náročnejšia je práve verifikácia. Z toho dôvodu boli vyvinuté metodológie, ktoré sa snažia minimalizovať čas a úsilie potrebné na vytváranie verifikačných prostredí poskytovaním návodu ako konštruovať znovupoužiteľné a spolupracujúce testbenche a verifikačné IP (VIP). Umožňujú to najmä prostredníctvom množiny základných tried. Tie uľahčujú prácu s prostredím a dátami, komunikáciu medzi procesmi, správu získaných údajov. Typickým reprezentantom je napríklad *Open Verification Methodology* [9] [1].

Open Verification Methodology (OVM) vznikla v roku 2007 ako výsledok spoločného úsilia Cadence Design System a Mentor Graphics o vytvorenie metodológie pre jazyk SystemVerilog spĺňajúcej požiadavky na interoperabilitu a otvorenosť. Objektovo orientovaný prístup k tvorbe verifikačných prostredí sa spolieha na sadu tried, ktoré definujú typy objektov slúžiacich ako stavebné bloky a metód pracujúcich s nimi. OVM ponúka presne takúto knižnicu tried (vrátane zdrojových kódov a dokumentácie) využiteľných pri konštrukcii generátorov stimulov, monitorov protokolov a kontrolórov výsledkov. Táto knižnica je napísaná s použitím štandardu IEEE 1800 SystemVerilog a nevyužíva žiadne ďalšie proprietárne rozšírenia.



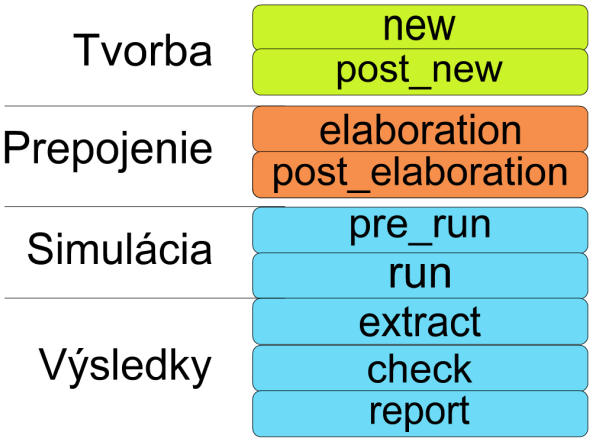
Obrázok 2.1: Vzťah medzi OVM a SystemVerilog

Ako vidno na obrázku 2.1, OVM knižnica je základ, nad ktorým je vystavané verifikačné prostredie. Dokumentácia metodológie poskytuje detaily ako konštruovať testbenche a znovupoužiteľné VIP. Ako už bolo spomínané, jadro OVM tvorí SystemVerilog. Ten však sám o sebe nie je dostačujúci. Bez stavebných blokov a metodológie popisujúcej ako ich

používať by nebolo možné zabezpečiť interoperabilitu.

Objektovo orientovaná povaha OVM umožňuje prispôsobenie testbenchu bez nutnosti zasahovať do zdrojových kódov knižnice, potrebné nové schopnosti sa zahrnú do tried odvodených. Preto sú tieto pôvodné triedy často nazývané "základné", aby vyjadrili tú skutočnosť, že poskytujú iba základnú množinu vlastností, ktorá môže byť ľubovoľne rozšírená.

Ďalšou významnou vlastnosťou je dynamická povaha vytváraných testbenchov. Komponenty testbenchu vznikajú a sú zostavované pri štarte simulácie, preto testbenche môžu byť modifikované priebežne. To poskytuje vysoký stupeň flexibility počas generovania testov. Rovnako umožňuje spustenie rôznych scenárov z toho istého základného testbenchu iba s minimálnou nutnosťou zmeniť kód. Tým vytvára OVM priestor pre znovupoužiteľnosť, či už v rámci projektu alebo medzi projektmi. Dynamická povaha vyžaduje, aby simulácia prebiehala jasne stanovenými fázami tvorby testbenchových komponentov a ich následného použitia. Na obrázku 2.2 sú zobrazené fázy simulácie, ako sú definované OVM. Všetky testbenche a VIP prechádzajú týmito fázami, aby sa zaručilo, že pri výskyte v rovnakom verifikačnom prostredí sú schopné spolupracovať. Fázy berú do úvahy dynamickú tvorbu komponentov, vzájomné prepojenie, proces simulácie a zaznamenanie výsledkov.



Obrázok 2.2: Simulačné fázy OVM

Verifikácie vyžadujú okrem vhodného jazyka aj jednoduchú tvorbu prostredí použitím stavebných blokov a správnej metodológie. Pokusy o vytvorenie takejto metodológie v minulosti z rôznych dôvodov zlyhávali. Niektoré boli príliš jednoduché aby mohli podporovať *constrained-random coverage-driven* testovanie, niektoré využívali proprietárne rozšírenia a iné boli viazané licenčnými podmienkami.

OVM ako priemyselne prvá otvorená verifikačná metodológia zabezpečujúca interoperabilitu bola rýchlo prijatá užívateľmi. Cadence a Mentor Graphics vytvorili knižnicu a metodológiu, ktorá pokrýva potreby sofistikovaných verifikátorov, ale zároveň podporuje aj jednoduché testbenche a testy písané menej skúsenými užívateľmi.

2.1 Verifikačný plán

Tradičný proces plánovania verifikácie zahŕňa určenie testov zameraných na špecifické vlastnosti dizajnu a popis konkrétnych stimulov aplikovaných na dizajn. Niekedy môžu byť testy

samokontrolné, vyhľadávajúce špecifické príznaky zlyhania na výstupe z dizajnu. Testy potom implementujú rôzni verifikátori. Tento spôsob verifikácie je podobný budovaniu celého dizajnu blok za blokom, pričom dúfame, že keď bloky spojíme, výsledok bude spĺňať všetky požiadavky.

Proces dizajnu je rozdelený do troch krokov: najprv požiadavky, potom architektúra a nakoniec detailná implementácia. Proces plánovania verifikácie by mal pozostávať z podobných krokov.

2.1.1 Verifikačné požiadavky

Definovanie verifikačných požiadaviek má za účel popísať vlastnosti, ktoré dokazujú, že dizajn spĺňa zamýšľanú funkciu. Požiadavky formujú základ, z ktorého vychádza zvyšok procesu plánovania verifikácie. Mali by byť identifikované hneď ako je to možné, najlepšie počas návrhu architektúry. Odporúča sa, aby na požiadavkách pracovali ľudia z projektu aj mimo projektu, dizajnéri, verifikátori a softvéroví inžinieri a zabezpečili tak definovanie požiadaviek z hardvérovej aj softvérovej perspektívy. Verifikačné požiadavky by mali zahŕňať definíciu toho,

- čo má dizajn robiť,
- čo nesmie dizajn robiť,
- aká funkcionalita nie je pokrytá verifikačným plánom.

Požiadavky by mali byť jednoznačne identifikované a tento identifikátor by sa nemal znovu použiť pre inú požiadavku, aj keď pôvodná už bola odstránená. Ďalej by sa mali odkazovať na špecifikáciu dizajnu, mali by byť kategorizované podľa dôležitosti a usporiadané v závislosti na iných požiadavkách. Zároveň vytvárajú základ pre funkčné pokrytie.

2.1.2 Požiadavky na verifikačné prostredie

Základný cieľ tohto kroku je definovať požiadavky na verifikačnú infraštruktúru, a tak zvýšiť pravdepodobnosť, že prostredie bude bez chýb. Ďalej pomáha identifikovať zdroje potrebné na implementáciu verifikačných požiadaviek.

Nie všetky časti dizajnu sú rovnaké. Niektoré zaobstarávajú kritické funkcie s nízkou schopnosťou kontrolovateľnosti, iné prevádzajú časovo náročné transformácie ľahko sledovateľné na výstupe. Niektoré verifikačné požiadavky teda jednoduchšie dosiahneme, ak rozdelíme dizajn na menšie časti. Tie poskytujú lepšie pozorovateľné a kontrolovateľné výstupy, zároveň však zvyšujú počet verifikačných prostredí, ktoré musíme vytvoriť. Je teda na verifikátorovi, aby zvážil, ktoré časti dizajnu testovať samostatne.

Každý nezávisle verifikovaný dizajn predstavuje množinu rozhraní, ktoré musia byť riadené alebo monitorované verifikačným prostredím. Ak sa na viacerých dizajnoch vyskytuje rovnaké rozhranie, môže byť riadené alebo monitorované tým istým verifikačným prvkom. To znižuje počet komponentov, ktoré musíme nanovo vytvoriť pri budovaní požadovaného verifikačného prostredia. Verifikačné požiadavky sa ale môžu u jednotlivých dizajnov (hoci používajú rovnaké rozhranie) líšiť, preto ak chceme vytvoriť znovupoužiteľné komponenty, musíme brať ohľad na všetky požiadavky.

Dizajn často podporuje viaceré konfigurácie. Vtedy treba určiť, ktoré z nich sa budú verifikovať, prípadne či sa budú vyberať náhodne. Náhodný výber musíme obmedziť príslušnými podmienkami, aby sme dostávali relevantné konfigurácie. Pomocou funkčného

pokrytia sledujeme, ktoré sa nám už podarilo zverifikovať. Niektoré aspekty konfigurácie, ako napríklad počet portov zariadenia, sú pevne stanovené v čase kompilácie. Tie už verifikačné prostredie počas simulácie nemá možnosť zmeniť. Preto sa využíva dvoj-priechodové náhodné generovanie dizajnu, kde prvý priechod slúži na výber konfigurácie, podľa ktorej sa dizajn skompiluje.

Verifikačné požiadavky popisujú, čo by mal dizajn robiť, keď pracuje správne. Pri testovaní vykonávame kontrolu očakávaného správania najmä pomocou samokontrolných mechanizmov. Samokontrolné mechanizmy odhaľujú odchýlky oproti očakávaným hodnotám. Čím jasnejšie sú príznaky zlyhania, tým jednoduchšie je overovať výstupy dizajnu. Samokontrolné mechanizmy vyberáme na základe očakávaných chýb. Niektoré chyby sú zrejmé už na úrovni signálov. Tie overujeme pomocou *assertions*. Chyby, ktoré sa dajú odhaliť akonáhle transakcia opustí dizajn, najľahšie zachytíme pomocou *scoreboardu*. Výkonnosť alebo spravodlivé prideľovanie zdieľaných prostriedkov kontrolujeme štatistickými analýzami. Tie pre nás zabezpečuje *offline checking mechanizmus*. Dostupnosť samokontrolného mechanizmu je daná dostupnosťou spôsobu predpovedať očakávané odpovede dizajnu. V prípade existencie referenčného alebo matematického modelu môže byť teda použitie offline checking mechanizmu efektívnejšie ako použitie scoreboardu.

Mnohé verifikačné požiadavky môžu byť splnené, iba ak sa dizajn dostane do určitého stavu. Požadovaný stav navodíme aplikovaním správnej sekvencie stimulov. Jednoduchým riešením je použitie riadených testov, avšak tomuto spôsobu (vyžadujúcemu zdĺhavé písanie testov) sa práve snažíme vyhnúť. Správny prístup je teda obmedzenie náhodného generátora tak, aby sme zvýšili pravdepodobnosť generovania potrebnej sekvencie stimulov.

Prvé testy, ktoré aplikujeme na dizajn, sú takzvané triviálne. Ich úlohou nie je naplniť verifikačné požiadavky, ale overiť základnú funkcionalitu. Vykonanie zápisového cyklu po čítaní alebo poslanie jedného paketu sú typické príklady takýchto testov. Triviálne testy nemusia byť riadené. Tak ako môžu overiť, že dizajn je živý, môžu slúžiť aj na otestovanie základnej funkcionality verifikačného prostredia.

2.1.3 Implementačný plán verifikácie

Hlavný cieľ implementačného plánu je zabezpečiť, že implementácia poskytne potrebné pokrytie dizajnu a jeho funkcionality. Vychádza z požiadaviek na verifikačné prostredie popísaných vyššie. S ohľadom na životný cyklus projektu by mal byť implementačný plán dokončený ešte pred začatím fázy písania kódu dizajnu.

Verifikačné požiadavky sú do implementácie zahrnuté prostredníctvom modelu funkčného pokrytia a umožňujú tak automaticky sledovať pokrok vo verifikácii. Na vytvorenie tohto modelu sa v SystemVerilogu používa kombinácia *covergroup* a *cover property*. Cover property sa využíva pri snímaní dát na úrovni signálov a dokáže pokryť iba jeden bod. Pre snímanie dát na vyššej úrovni je určené covergroup, ktoré umožňujú pokryť viacero bodov.

Produkovanie správnych sekvencií stimulov vyžaduje patričné obmedzenie náhodného generátora. Komplexnosť obmedzení však stúpa spoločne so špecifickosťou požadovaných stimulov. Ak sa nám stále nedarí produkovať žiadané stimuly, jednoduchším riešením, ako naďalej komplikovať generátor, je vytvoriť sadu riadených stimulov. Tie potom môžu byť náhodne vkladané medzi ostatné stimuly získavané generátorom.

Posledná otázka, ktorú si pri vytváraní implementačného plánu kladieme, je stanovenie podmienky na ukončenie testu. Simulovanie dopredu určeného časového úseku nemusí zaručiť, že dizajn sa dostal do žiadaného stavu. Podmienkami pre ukončenie testu môže byť napríklad určitý počet chýb, ktoré sa vyskytli, dosiahnutie daného stupňa pokrytia

alebo prechod dizajnu do stavu nečinnosti.

Rovnako je dôležité, aby sme boli schopní zopakovať ktorýkoľvek test aplikovaný na dizajn. Často nám poslúži vhodný skript, ktorý sa postará o zaznamenanie verzií zdrojových súborov, knižníc, konfigurácie a počiatočného nastavenia generátora.

2.2 Verifikačné prostredie – Testbench

Testbench je kompletné verifikačné prostredie, ktorého úlohou je overiť správnosť testovaného dizajnu (design under test – DUT). To je realizované zabezpečením nasledujúcich krokov:

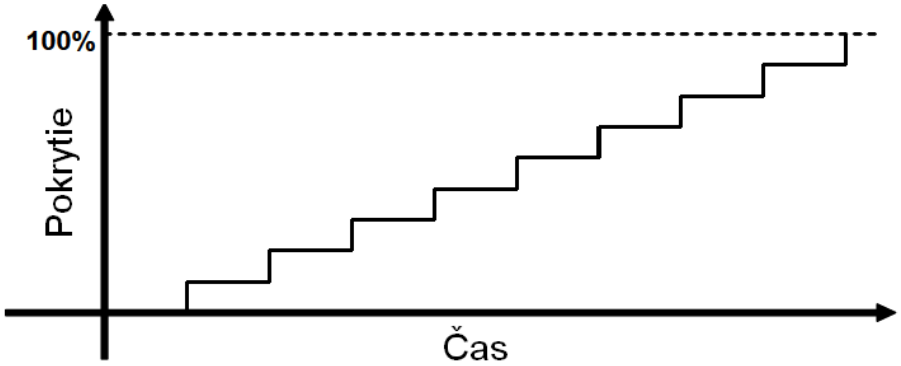
- generovaním stimulov,
- aplikovaním stimulov na verifikovaný dizajn,
- odchytávaním odpovedí,
- kontrolou správnosti,
- meraním progresu s ohľadom na ciele verifikácie.

Niektoré z týchto krokov sú zabezpečené automaticky testbenchom, iné musí manuálne overiť verifikátor. Toto rozdelenie závisí na použitej metodológii.

2.2.1 Riadené testovanie

Tradičný spôsob testovania dizajnu je použitie riadených testov (directed tests). Verifikátor si preštuduje hardvérovú špecifikáciu a napíše verifikačný plán obsahujúci zoznam testov, každý zameraný na určitú sadu vlastností. Podľa zoznamu vytvára testovacie vektory, postupne ich aplikuje na DUT a ručne kontroluje výsledky, či zodpovedajú očakávaniam. Ak test prebehol správne, vyškrtne ho zo zoznamu a posunie sa na ďalší.

Tento prístup poskytuje okamžité výsledky, keďže nie je potrebné vytvárať komplexnú infraštruktúru a môže sa začať priamo s testovaním. Obrázok 2.3 ukazuje, ako riadené testovanie postupne pokrýva všetky vlastnosti vo verifikačnom pláne. Každý test je zameraný na špecifickú časť dizajnu. Ak máme dostatok času, môžme napísať potrebné množstvo testov a dosiahnuť tak 100% pokrytie verifikačného plánu.



Obrázok 2.3: Progres riadeného testovania

Ako vidno, sklon krivky je konštantný, a teda ak sa zložitosť DUT zdvojnásobí, jej verifikácia zaberie dvakrát viac času alebo si vyžiada dvakrát toľko ľudí. To je takmer vždy neprijateľné, preto sa využívajú metodológie, ktoré dosiahnu požadované pokrytie rýchlejšie.

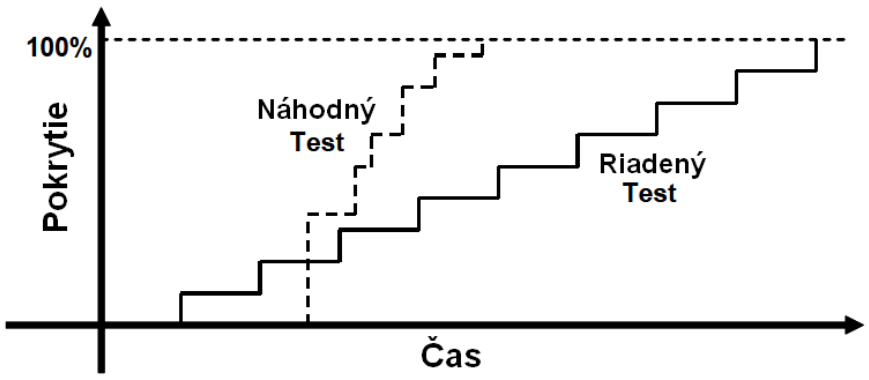
2.2.2 Constrained-random testovanie

Generovanie náhodných stimulov je kľúčové pre verifikáciu komplexných dizajnov. Využívajú sa spoločne s ďalšími princípmi, čím vytvárajú základ metodológie:

- funkčné pokrytie (functional coverage),
- úrovňové testbenche,
- spoločný testbench pre všetky testy,
- kód špecifický pre určitý test nezahrnúť do testbenchu.

Riadené testy nájdu chyby, ktoré sa v DUT očakávajú, zatiaľ čo náhodné testy môžu odkryť chyby, ktoré nikto nepredpokladal. Pri použití náhodných stimulov využívame na meranie progresu funkčné pokrytie, očakávané výsledky nie sú kontrolované ručne, ale automaticky sa predpovedajú výsledky a porovnávajú. Vytvorenie takejto štruktúry vyžaduje značné množstvo práce. Úrovňové testbenche pomáhajú zvládnuť túto komplexnosť rozdelením problému na menšie časti.

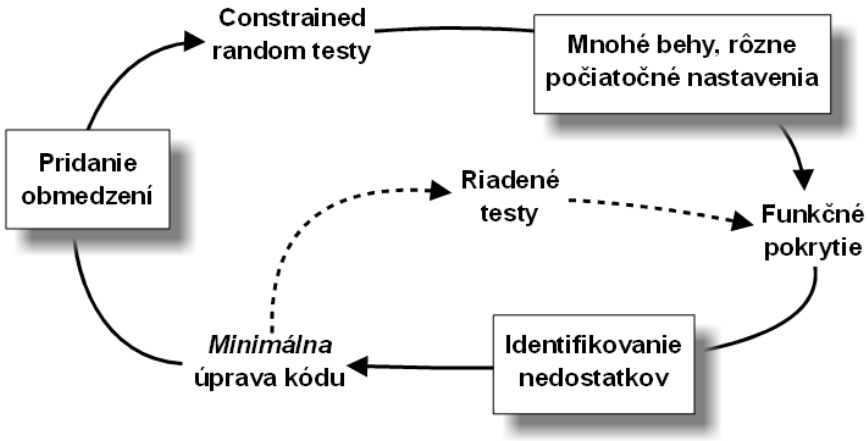
Tento štýl vytvárania verifikačných prostredí zaberá viac času ako tradičné riadené testovanie, spôsobuje tak oneskorenie pred spustením prvých testov. To môže z počiatku pôsobiť skľučujúco, ale všetko je vynahradené následným ziskom. Každý test, ktorý vytvoríme, využíva tento spoločný testbench, na rozdiel od riadených testov, ktoré sú písané od začiatku. Každý náhodný test obsahuje niekoľko riadkov kódu, ktoré žiadaným spôsobom obmedzujú generované stimuly. Výsledkom je, že náš jediný constrained-random testbench nájde chyby rýchlejšie ako početné riadené testy, čo naznačuje obrázok 2.4.



Obrázok 2.4: Progres constrained-random testovania

Ak rýchlosť pokrývania začína klesať, môžeme vytvoriť nové obmedzenia náhodných stimulov a preskúmať tak zvyšné oblasti. Na odhalenie posledných niekoľko chýb bude zrejme nutné použiť riadené testovanie, ale drvivá väčšina sa nájde už pomocou náhodných testov.

Obrázok 2.5 ukazuje spôsob, ako dosiahnuť úplné pokrytie. Začíname constrained-random testami. Spúšťame ich s rôznymi počiatočnými nastaveniami. Sledujeme funkčné pokrytie a minimálne upravujeme kód, najčastejšie iba zmenou obmedzení. Pre zvyšných zopár vlastností, u ktorých je veľmi nepravdepodobné, že budú pokryté náhodnými testami, napíšeme riadené testy.

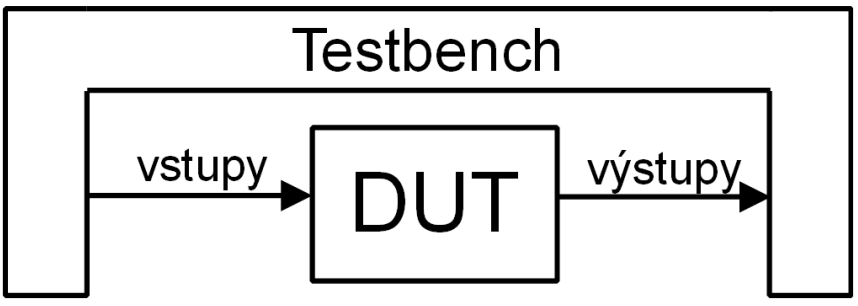


Obrázok 2.5: Konvergencia pokrytia

2.2.3 Architektúra testbenchu

Testbench obopína DUT podobne ako hardvérový tester pripojený k fyzickému čipu. Oba, testbench aj tester, poskytujú stimuly a zachytávajú odpovede. Rozdiel medzi nimi je, že testbench pracuje cez niekoľko úrovní abstrakcie, vytvára transakcie a sekvencie, ktoré sú transformované na bitové vektory. Tester pracuje iba na úrovni bitov.

Testbench pozostáva z funkčných modelov rozhrania (Bus functional model – BFM), ktoré vytvárajú jednotlivé komponenty verifikačného prostredia. Pre testovaný dizajn vyzerajú ako skutočné komponenty. Ak sa napríklad DUT pripája k USB, PCI a SPI rozhraniu, musíme vytvoriť vo verifikačnom prostredí odpovedajúce komponenty, ktoré dokážu generovať stimuly a zachytávať odozvy.

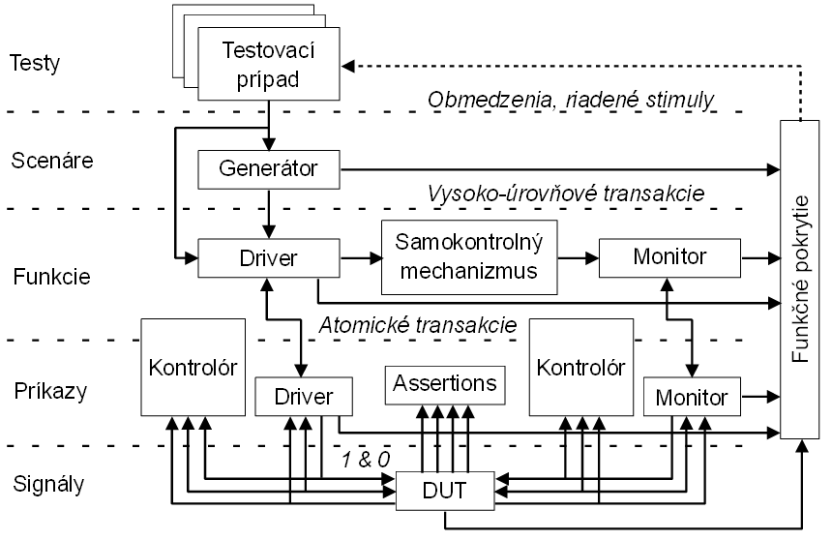


Obrázok 2.6: Verifikačné prostredie

Úrovňový testbench

Základným konceptom každej modernej verifikačnej metodológie je úrovňový testbench. Hoci sa zdá, že týmto sa stáva verifikačné prostredie zložitejšie, v skutočnosti verifikátorovi uľahčuje úlohu rozdelením kódu na menšie časti, ktoré môžu byť vyvíjané nezávisle.

Každá úroveň poskytuje množinu služieb nadradenej vrstve, čím vytvára abstrakciu nad nižšími vrstvami.



Obrázok 2.7: Úrovňový testbench

Aj keď obrázok 2.7 ukazuje interakciu testovacích prípadov iba s najvyššou vrstvou verifikačného prostredia, tie môžu priamo ovplyvňovať ktorýkoľvek komponent prostredia, aby splnili svoj cieľ. Testovacie prípady sú implementované ako kombinácia obmedzení na generátory, definície náhodných scenárov, synchronizačných mechanizmov medzi vybavovacími jednotkami (transactor), vkladaných chýb a riadených stimulov. Pretože po verifikačnom prostredí sa požaduje, aby bolo schopné, bez modifikácií, aplikovať všetky potrebné testovacie prípady, musí byť zostavené zo starostlivo navrhnutých znovupoužiteľných komponentov.

Úroveň signálov

Táto najnižšia vrstva pozostáva z testovaného dizajnu a signálov na pripojenie k testbenchu. Poskytuje abstrakciu nad názvami jednotlivých pinov a tak umožňuje použitie verifikačných komponentov s rôznymi DUT bez nutnosti modifikácie. Táto abstrakcia je dostupná a môže byť použitá všetkými vrstvami a testovacími prípadmi, ak je to potrebné. Avšak verifikačné prostredia by mali byť vytvárané tak, aby vyššie vrstvy využívali služieb nižších vrstiev a pristupovali priamo k signálom iba ak je to naozaj nevyhnutné.

Úroveň príkazov

Tradičné testovacie prípady pozostávajúce z transakcií písané vo Verilogu alebo VHDL sú typicky implementované na tejto úrovni.

Úroveň príkazov pozostáva z BFM, driverov, monitorov a kontorlórov spojených s rôznymi rozhraniami a protokolmi fyzickej úrovne, ktoré sa vyskytujú v DUT. Zároveň dispo-

nuje samokontrolným mechanizmom v podobe assertions. Vrstva poskytuje nízkoúrovňové transakčné rozhranie k testovanému dizajnu, transakcie sú definované ako atomické prenosy dát alebo príkazové operácie na rozhraní, napríklad zápis do registru, prenos ethernetového rámca či načítanie inštrukcie.

Driver dodáva stimuly do DUT. *Proaktívny* driver riadi inicializáciu a zápis transakcií. Kedykoľvek vyššie vrstvy verifikačného prostredia poskytnú novú transakciu, driver ju ihneď vykoná na fyzickom rozhraní. *Reaktívny* driver neriadi inicializáciu ani zápis transakcií, ale môže mať na starosti časovanie ich vykonania vkladaním čakacích stavov na rozhranie. Ďalšou úlohou reaktívneho drivera je poskytnutie potrebných dát na dokončenie transakcie, ktorú započne DUT. Príkladom je BFM programovej pamäte. Dizajn iniciuje načítanie ďalšej inštrukcie a BFM podskytne nové dáta reprezentujúce zakódovanú inštrukciu.

Monitor na základe signálov z DUT prijíma dáta a príkazy a poskytuje ich ďalej vyšším vrstvám. *Reaktívny* monitor generuje signály potrebné na povolenie prenosu alebo úspešné dokončenie transakcie v prípade handshakingu, *pasívny* monitor iba jednoducho sleduje signály bez akéhokoľvek zásahu. Je vhodný na monitorovanie transakcií na rozhraniach medzi dvoma blokmi DUT.

Funkčná úroveň

Funkčná úroveň poskytuje nevyhnutnú vrstvu abstrakcie na spracovanie aplikačných transakcií a overenie správnosti testovaného dizajnu. Samokontrolný mechanizmus overuje správnosť odpovedí DUT založených na konfigurácii a aplikovaných stimuloch.

Transakcie na funkčnej úrovni nemusia odpovedať jedna k jednej fyzickým transakciám. Funkčné transakcie sú abstrakciou vysokoúrovňových operácií vykonávaných značnou časťou alebo celou DUT. Jedna funkčná transakcia môže predstavovať vykonanie desiatok transakcií na úrovni príkazov na rôznych rozhraniach.

Komponenty pracujúce na tejto vrstve sa podľa [3] nazývajú vybavovacie jednotky (transactors). Transaktory môžu byť proaktívne, reaktívne alebo pasívne. *Proaktívne* transaktory riadia inicializáciu a druh transakcie a typicky poskytujú niektoré alebo všetky dáta vyžadované transakciou. *Reaktívne* transaktory sú zodpovedné iba za dokončenie transakcie poskytnutím potrebných dát alebo handshakingom. *Pasívne* transaktory monitorujú transakcie na rozhraní.

Úroveň scenára

Scenár je postupnosť transakcií s určitými vzťahmi. Každý scenár predstavuje sekvenciu individuálnych transakcií s cieľom pokryť určitý okrajový prípad použitia. Napríklad scenár prenosu na ethernete bude postupnosť rámcov so špecifickou hustotou – určitú dobu bude ethernetová linka zaneprázdnená vysielaním/prijímaním, inokedy bude nevyužitá. Generátory generujú scenáre v náhodnom poradí a poskytujú tok transakcií, ktorý odpovedá generovanému scenáru. Generátory sú riadené určitým testovacím prípadom.

Táto vrstva môže byť z časti alebo úplne obídená testovacou vrstvou nad ňou v závislosti na požiadavkach testovacieho prípadu. Musíme mať teda možnosť generátory vypnúť, či už na začiatku alebo uprostred simulácie a umožniť tak aplikovanie riadených testov. Generátor musí takisto disponovať možnosťou znovuspustenia.

Testovacia úroveň

Testovacie prípady obsahujú kombináciu obmedzení na generátory, definície nových náhodných scenárov, synchronizáciu rôznych transaktorov a riadené stimuly.

Táto vrstva môže tiež poskytovať prídavnú kontrolu špecifickú pre daný testovací prípad, ktorú neposkytuje funkčná vrstva. Jedná sa napríklad o situácie, kde správnosť závisí na časovaní s ohľadom na určitú synchronizačnú udalosť vyskytujúcu sa v testovacom prípade.

Či je potrebné, aby sa všetky tieto vrstvy nachádzali v našom testbenchi, závisí na DUT. Komplikovaný dizajn vyžaduje sofistikovaný testbench, pre jednoduchšie dizajny býva úroveň scenárov taká jednoduchá, že ju môžme spojiť s funkčnou vrstvou.

Takisto sa môže vyskytnúť situácia, že budeme potrebovať viac úrovní. Napríklad pre TCP tok zabalený do IP paketov a posielaný cez ethernet vytvoríme na funkčnej úrovni niekoľko podúrovní, ktoré zabezpečia zapuzdrenie a kontrolu dát, paketov a rámcov.

Kontrola odpovedí

Z verifikačných požiadaviek môžeme odvodiť chyby, ktoré by malo naše verifikačné prostredie odhaliť. Na to potrebujeme mechanizmy, ktoré budú predikovať očakávané odpovede a porovnávať ich s tými skutočne obdržanými.

V tradičnom riadenom testovaní, pretože stimuly aj funkcionalitu dizajnu poznáme, môžeme dopredu odvodiť očakávané odpovede a natvrdo ich zakomponovať do testu. Pri náhodných stimuloch poznáme iba funkcionalitu komponentu, nie však samotný stimul. Predpokladaná odpoveď teda musí byť priebežne počítaná na základe konfigurácie a funkcionality dizajnu a následne porovnaná s odpoveďou od DUT.

Je dôležité si uvedomiť, že iba štruktúra realizujúca kontrolu odpovedí je schopná odhaliť chybu dizajnu. Ak táto štruktúra priamo nekontroluje určitý príznak zlyhania, chyba zostane neodhalená.

Vstavané monitory

Vo všeobecnom ponímaní sa odpovede získavajú na vonkajších rozhraniach testovaného dizajnu. Avšak obmedzenie odpovedí iba na externé rozhrania môže zapríčiniť, že niektoré symptómy zlyhania bude ťažké odhaliť. Ak verifikačné prostredie nemá dostatočný prehľad o DUT, dokázanie korektného správania sa vnútornej jednotky môže zabrať príliš veľa času.

Transaktory nemusia byť obmedzené iba na vonkajšie rozhrania. Monitormi môžeme zrkadliť činnosť alebo dokonca nahradiť vnútornú jednotku dizajnu. Rozhranie vstavaného monitora zostáva dostupné z vonku a poskytuje tak prístup k rozhraniu jednotky.

Assertions

Termín assertion (tvrdenie, výrok) znamená výraz, ktorý je pravdivý. Z pohľadu verifikácie, assertion je výraz popisujúci očakávané správanie. Akýkoľvek rozpor v pozorovanom správaní vedie k chybe. Z tejto definície vyplýva, že celý testbench je jeden veľký assertion – je to výraz reprezentujúci očakávané správanie celého dizajnu. Vo verifikáciách sa však assertion používa na označenie vlastnosti použitím *temporálnych výrazov*.

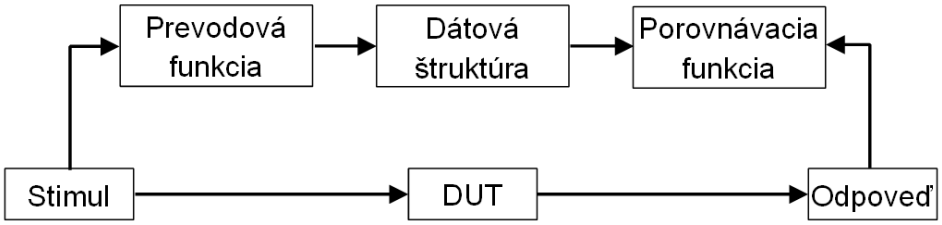
Použitie assertions na detekciu chýb je veľmi efektívne, pretože chyby sú reportované blízko (časovo aj priestorovo) udalosti, ktorá ich spôsobila. Napriek ich efektívnosti, assertions sú limitované typom vlastností, ktoré môžu byť vyjadrené temporálnou logikou.

Niektoré výrazy očakávaného správania dizajnu sa nedajú špecifikovať inak než procedurálnym kódom. Spolu s assertions potom spolupracujú pri odhaľovaní chýb. Napríklad pri kontrole protokolu sa pomocou assertions popíšu nízkoúrovňové vlastnosti ohľadne signálov a procedurálnym kódom vysokoúrovňové, zamerané na transakcie.

Asserions fungujú dobre pri verifikácii vzťahov medzi signálmi. Dokážu efektívne detekovať chyby v handshakingu, prechodoch medzi stavmi a pravidlách protokolu na fyzickej úrovni. Taktiež dokážu odhaliť neočakávané alebo nežiadúce stavy na najnižšej vrstve. Na druhú stranu, assertions nie sú vhodné na detekciu chýb, ktoré vyžadujú skladovanie veľkého množstva dát (odhalenie zmien v dátach, správne radenie paketov na základe ich priority, ...) alebo komplexné výpočty (kontrola CRC, ...).

Scoreboarding

Scoreboard sa používa na dynamickú predikciu odpovedí dizajnu. Ako vidno na obrázku 2.8, stimuly aplikované na DUT sú zároveň poskytnuté prevodovej funkcii. Prevodová funkcia vykonáva všetky potrebné transformácie a z pôvodného stimulu tak vytvorí očakávanú odpoveď, ktorú uloží do vhodnej dátovej štruktúry. Odpoveď získaná od DUT je preposlaná porovnávacej funkcii, ktorá overí, či sa zhoduje s očakávanou.



Obrázok 2.8: Scoreboarding

Prevodová funkcia pracuje na transakčnej úrovni a zmeny obyčajne vykonáva v nulovom čase. Dátová štruktúra uchováva očakávané odpovede, dokým môžu byť porovnané s výstupom z DUT. Vyhľadávanie v dátovej štruktúre a porovnanie získanej odpovede s očakávanou má na starosti porovnávacia funkcia. Tá, spolu s dátovou štruktúrou, umožňuje niektoré prípustné nezrovnalosti ako iné poradie alebo oneskorenie odpovedí z dizajnu. Prevodová funkcia a dátová štruktúra sú obyčajne konfigurovateľné, aby odpovedali aktuálnemu nastaveniu DUT.

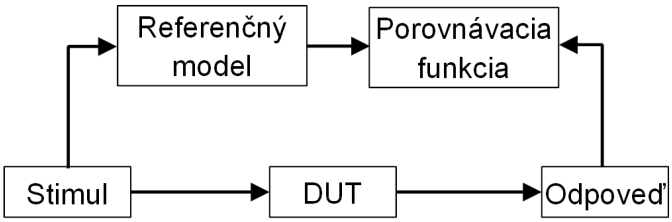
Scoreboarding je vhodný na overenie integrity výstupných dát. Dokáže efektívne detekovať chyby vo výpočtoch, transformácii a radení. Takisto ľahko zistí chýbajúce alebo nežiadúce dáta. Na druhú stranu, scoreboarding nie je príliš vhodný na detekciu chýb, ktorých symptómy nie sú pozorovateľné z jedinej odpovede. Spravodlivé prideľovanie prostriedkov alebo celkovú výkonnosť dizajnu overí iba ťažko.

Referenčný model

Referenčný model, podobne ako scoreboard, sa používa na dynamickú predikciu odpovedí dizajnu. Stimuly aplikované na dizajn sú súčasne poskytované referenčnému modelu. Výstup modelu je porovnávaný s odpoveďami získanými z DUT, ako ilustruje obrázok 2.9.

Referenčný model má rovnaké schopnosti ako scoreboard. Na rozdiel od scoreboardu, porovnávacia funkcia pracuje priamo s výstupom modelu. Referenčný model preto musí

produkovať výstup v rovnakom poradí ako samotný dizajn. Nie je však potrebné produkovať ho s rovnakým oneskorením ako DUT, s touto nezrovnalosťou si poradí porovnávacia funkcia. Porovnávanie získaných odpovedí s predikovanými sa deje na transakčnej úrovni.

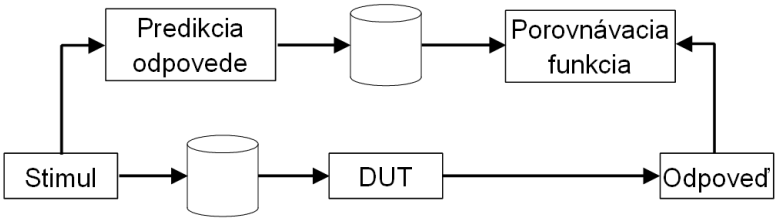


Obrázok 2.9: Referenčný model

Použitie referenčného modelu závisí silne na jeho dostupnosti. Ak je k dispozícii, mal by byť použitý, ak nie je, nahradíme ho technikou scoreboardingu.

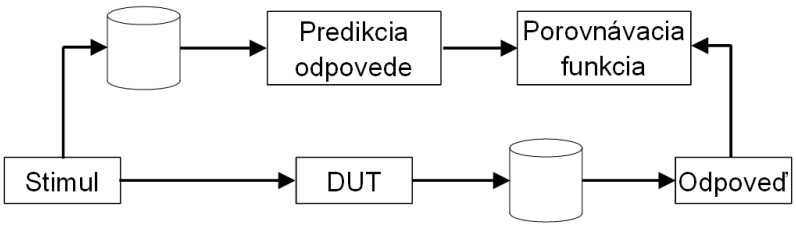
Offline kontrola

Offline kontrola sa používa na predikovanie odpovedí dizajnu pred alebo po samotnej simulácii. Ako znázorňuje obrázok 2.10, v pre-simulačnej predikcii offline kontrolór produkuje popis očakávaných odpovedí, ktoré sú behom simulácie dynamicky porovnávané s odpoveďami z DUT.



Obrázok 2.10: Predsimulačná offline kontrola

V post-simulačnej predikcii, znázornenej na obrázku 2.11, zaznamenané stimuly a odpovede dizajnu sú porovnávané s výsledkami offline kontrolóra. V oboch prípadoch môžu byť odpovede overované s rôznym stupňom detailnosti.



Obrázok 2.11: Posimulačná offline kontrola

Použitie predsimulácie s dynamickou kontrolou odpovedí umožňuje odhaliť nezrovnalosti, keď je DUT v stave alebo blízko stavu, v ktorom k chybe došlo. Neposkytuje však

možnosť generovať stimuly o ohľadom na stav dizajnu a ten sa teda nedostane do všetkých prípustných stavov.

Offline checking sa hodí na overovanie integrity dát na základe systémovej špecifikácie alebo matematického modelu. Efektívne odhalí chyby vo výpočtoch, transformácii dát a usporiadaní, detekuje chýbajúce a nežiadúce dáta. Posimulačná offline kontrola je takisto vhodná na zistenie chýb, ktoré sa neprejavia v rámci jedinej odpovede. Spravodlivé prideľovanie prostriedkov alebo rozhodovanie na základe priority overí vykonaním štatistickej analýzy nad zaznamenanými odpoveďami.

2.3 SystemVerilog

Koncom deväťdesiatych rokov sa Verilog HDL stal najrozšírenejšie používaným jazykom pre popis hardvéru (Hardware Definition Language – HDL). Avšak prvé dve verzie špecifikované organizáciou IEEE (1364-1995 a 1364-2001) ponúkali iba malé možnosti pre tvorbu testov. Keď dizajn prerástol verifikačné schopnosti Verilogu, vznikali komerčné riešenia v podobe OpenVera a *e* HVL (Hardware Verification Language – jazyk na verifikáciu hardvéru). Spoločnosti, ktoré nechceli platiť za tieto nástroje, strávili stovky človeko-rokov vytváraním vlastných nástrojov.

Táto kríza viedla k vytvoreniu konzorcia Accellera, zloženého z EDA spoločností a užívateľov, ktorí chceli vytvoriť novú generáciu Verilogu. Cieľ sa podarilo naplniť v novembri 2005 prijatím IEEE štandardu 1800-2005 [4] pre *SystemVerilog* [10] [11].

Verifikácia je všeobecne videná ako zásadne odlišná činnosť od návrhu. To viedlo k vytvoreniu jazykov úzko zameraných na verifikáciu a rozdeneniu vývojárov na dve skupiny. Špecializácia navyše viedla ku vzniku komunikačných problémov. SystemVerilog rieši tento problém poskytnutím jednotnej syntaxe a sémantiky pre dizajn aj verifikáciu. Aj keď návrhár nie je schopný napísať objektovo-orientované verifikačné prostredie, je preňho dostatočne pochopiteľné pri čítaní, čo umožňuje dizajnérovi a verifikátorovi pracovať spolu pri identifikovaní a opravení problému. Zahrnutie dizajnu, testbenchu a assertions do jedného jazyka poskytuje verifikačnému prostrediu jednoduchý prístup k všetkým častiam dizajnu bez použitia špeciálneho API.

2.3.1 HDL oblasť

SystemVerilog je kombinovaný jazyk pre popis a verifikáciu hardvéru. HDL časť je založená na Verilogu, HVL vychádza z jazyka OpenVera. V oblasti dizajnu poskytuje SystemVerilog niekoľko nových dátových typov ako viacrozmerné zhustené polia (multidimensional packed arrays) alebo dvoj-stavové celočíselné typy `bit`, `byte`, `shortint`, `int` a `longint`. Tie nepodporujú metahodnoty X a Z, práca s týmito typmi môže viesť k rýchlejšej simulácii. Štruktúry a únie sa viac pripodobnili tým v jazyku C, pridaním atribútu budú zaberať v pamäti súvislý blok, čo umožňuje prácu ako so zhusteným poľom. Atribúty podporujú aj konštrukcie `if/case`. Ich aplikovaním zabezpečíme prioritné alebo paralelné vyhodnocovanie jednotlivých položiek.

2.3.2 HVL oblasť

V oblasti verifikácie predstavuje SystemVerilog takisto niekoľko nových dátových typov, najmä s premenlivou dĺžkou: reťazec, dynamické pole, asociatívne pole a front, ktorý ponúka väčšinu funkcionality C++ STL deque.

SystemVerilog poskytuje objektovo-orientovaný programovací model. Triedy podporujú jednoduchú dedičnosť, neexistuje možnosť zhody triedy s viacerými funkčnými rozhraniami, ako je tomu v Jave. Triedy môžu byť parametrizované, ponúkajú tak základnú funkčnosť C++ šablón. Funkcionálne šablóny a špecializáciu šablón však nepodporuje. Polymorfizmus je podobný tomu v C++, využíva virtuálne funkcie. Zapúzdrenie a skrývanie atribútov sa deje pomocou kľúčových slov `local` a `protected`, implicitne sú všetky atribúty a metódy verejné. Jazyk využíva garbage collection, takže nedisponuje prostriedkami na implicitné zrušenie inštancie triedy.

Celočíselným veličinám môžu byť priradené náhodné hodnoty na základe množiny obmedzení. To umožňuje vytvárať constrained-random generované stimuly. Obmedzenia môžu byť ľubovoľne komplexné, obsahujúce vzťahy medzi premennými, implikácie a iterácie.

SystemVerilog má vlastný jazyk špecifikujúci assertions – *SystemVerilog Assertions*. Je podobný jazyku Property specification language. Assertions pozostávajú zo sekvencií (sequences) a vlastností (properties). Vlastnosti sú nadmnožinou sekvencií, každá sekvencia môže byť použitá, ako keby bola vlastnosť, hoci to nie je typicky výhodné. Sekvencie pozostávajú z pravdivostných výrazov rozšírených o temporálne operátory, opakovania a rôzne konjunkcie. To umožňuje verifikátorovi vyjadriť zložité vzťahy medzi komponentmi dizajnu. Assertions sa neustále pokúšajú vyhodnotiť sekvenciu alebo vlastnosť. Assertion zlyhá ak zlyhá vlastnosť.

Okrem assertions, SystemVerilog podporuje predpoklady (assumptions) a pokrytie vlastností. Predpoklad vytvára podmienku, ktorú dokazovací nástroj formálnej logiky musí prijať za pravdivú. Assertions špecifikujú vlastnosti, ktoré musia byť dokázané ako pravdivé. V simuláciách sú assertions aj predpoklady overované voči stimulom. Pokrytie vlastností umožňuje verifikátorovi overiť, že assertions správne monitorujú dizajn.

Pri určovaní, či dizajn bol vystavený dostatočne rôznorodým stimulom sa využíva pokrytie (coverage). Na rozdiel od pokrytia kódu (code coverage), ktoré zaručuje, že všetky riadky kódu boli vykonané, funkčné pokrytie zistí, či boli preskúmané požadované okrajové prípady. SystemVerilogový `covergroup` vytvára histogram hodnôt danej premennej. Umožňuje takisto krížové pokrytia (cross coverage) viacerých premenných, kde histogram reprezentuje karteziánsky súčin premenných.

Komplexné testovacie prostredie pozostáva zo znovupoužiteľných verifikačných komponentov, ktoré navzájom komunikujú. Vo Verilogu bola medzivláknová synchronizácia ponechaná plne na programátora, iba kľúčové slovo `event` umožňovalo navzájom spúšťať jednotlivé bloky procedurálnych výrazov. Jazyk SystemVerilog disponuje ďalšími dvoma prostriedkami pre medzivláknovú synchronizáciu, a tým je `mailbox` a `semaphore`. Základom mailboxu je FIFO front. FIFO môže byť parametrizované, čo zapríčiní, že iba objekty určitého typu môžu byť prostredníctvom neho predávané. Typicky sú týmito objektami inštancie tried reprezentujúcich transakcie – pakety alebo rámce spracovávané dizajnom.

Kapitola 3

Protokol FrameLink a nástroje pre prácu s ním

FrameLink je protokol pre prenos dát vo forme paketov v rámci dizajnov vytvorených na projekte Liberouter [7]. Nahradzuje predtým používaný command protokol, ktorý je pri vyšších dátových šírkach značne neefektívny. FrameLink je inšpirovaný protokolom LocalLink od Xilinxu [8], ale obsahuje niektoré závažné zmeny.

3.1 Špecifikácia protokolu FrameLink

3.1.1 Rozhranie protokolu

Rozhranie protokolu FrameLink tvoria nasledujúce signály:

- DATA – prenášané dáta o šírke N*8 bitov
- REM – signál určuje, koľko bajtov z dátovej cesty je platných. Signál je binárne zakódovaný, takže jeho šírka je $log_2(N)$. Signál je platný, pokiaľ je aktívny signál EOP_N – tzn. je koniec bloku a zaujíma nás, koľko bajtov v tomto poslednom slove je platných. Pokiaľ končí blok dát a REM je "00", tak je platný iba nultý bajt, "01"je platný nultý a prvý bajt, "11"sú platné všetky bajty.
- SOF_N – start of frame, aktívny v nule. Ohraničuje začiatok celého paketu, paket sa teda môže skladať z viacerých častí, ktoré sú ohraničené pomocou SOP_N a EOP_N. Zároveň so SOF_N je vždy nastavený SOP_N prvej časti paketu.
- EOF_N – end of frame, aktívny v nule. Ukončuje paket, môže sa vyskytovať spoločne so SOF_N (vtedy má paket menej než N bajtov). Zároveň s ním je vždy nastavený EOP_N poslednej časti paketu.
- SOP_N – start of part, aktívny v nule. Ohraničuje začiatok časti paketu. Tých môže byť v pakete ľubovoľné množstvo, typicky sú to však dve alebo tri.
- EOP_N – end of part, aktívny v nule. Ukončuje časť paketu, ohraničuje platnosť signálu REM. Môže sa vyskytovať zároveň so SOP_N, pokiaľ je daná časť menšia ako dátová šírka.

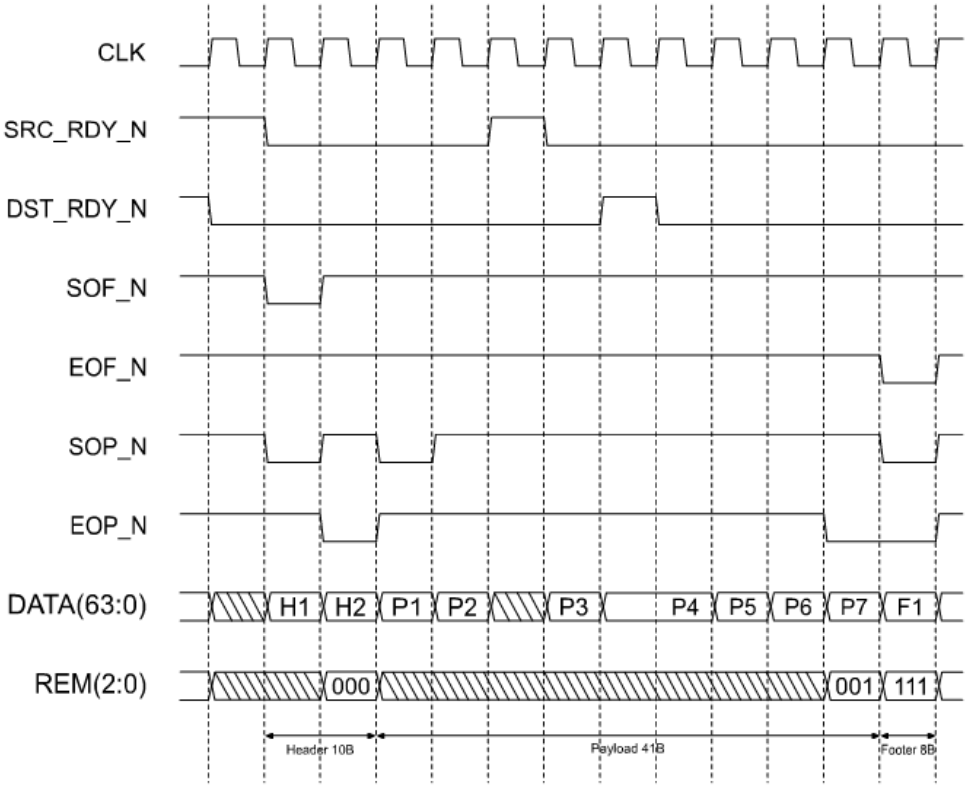
- SRC_RDY_N – source ready, aktívny v nule. Zariadenie, ktoré vysiela dáta, týmto signalizuje pripravenosť odosielať (má nejaké dáta na odoslanie). Dáta sú pripravené na výstupe. Takisto tento signál určuje platnosť všetkých ostatných signálov (riadiace signály, REM).
- DST_RDY_N – destination ready, aktívny v nule. Zariadenie prijímajúce dáta týmto signalizuje schopnosť prijať dáta.

3.1.2 Štruktúra paketu

Celý paket je ohraničený signálmi SOF_N a EOF_N. Skladá sa z ľubovoľného počtu častí ohraničených signálmi SOP_N a EOP_N. Každá časť je zarovnaná, teda pri SOP_N je vždy nultý bajt na nultej pozícii. V jednom slove dát sa preto nemôžu prenášať dve rôzne časti, ako je tomu v LocalLinku.

Paket sa bude typicky skladať z troch častí, ktorými budú hlavička (header), dáta (payload) a päta (footer). Ktorákoľvek časť môže byť vynechaná a vznikne tak napríklad iba header a payload. Počet častí je možné takisto zvýšiť, ale takéto použitie sa vo všeobecnosti nepredpokladá.

3.1.3 Prenos dát



Obrázok 3.1: Komunikácia protokolom FrameLink

Prenos je riadený signálmi SRC_RDY_N a DST_RDY_N, ktoré označujú pripravenosť

zariadenia prijímať/vysielať dáta. Vlastný prenos dát tak prebieha iba ak sú oba signály aktívne. U zariadenia, ktoré dáta odosiela, sa naviac predpokladá, že pokiaľ nejaké dáta na odoslanie bude mať, tak ich samo pripraví na výstup, ale na ďalšie dáta prejde až po potvrdení pripravenosti prijímať druhou stranou. Príklad prenosu paketu, ktorý sa skladá z troch častí, je na obrázku 3.1.

3.2 Nástroje pre protokol FrameLink

FrameLinkové nástroje sú charakteristické generickou šírkou vstupného a výstupného dátového portu. V prípade, že počet vstupných/výstupných rozhraní je iný ako jeden, dá sa takisto zvoliť príslušným generikom.

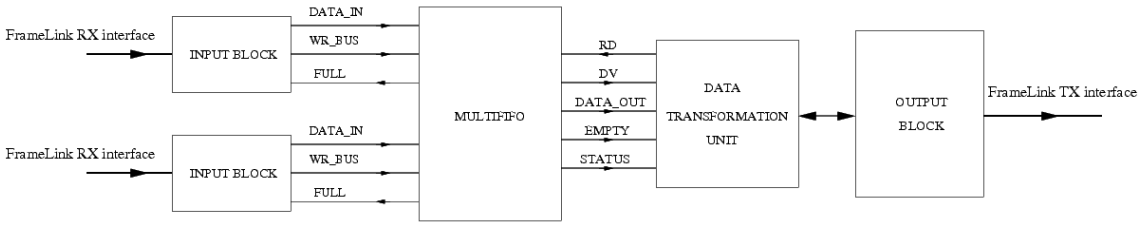
Transformer

Transformer slúži na prekonvertovanie vstupnej dátovej šírky na inú.

Binder

Binder spája niekoľko vstupných dátových tokov do jedného výstupného. Využíva sa v prípadoch, keď máme v dizajne jeden komponent s vysokou priepustnosťou a niekoľko s nižšou priepustnosťou. Uchováva dáta z N vstupných rozhraní o šírke DATA_WIDTH, ktoré potom získavame na výstupnom rozhraní šírky maximálne N*DATA_WIDTH. Poradie paketov závisí na zvolenom spôsobe výberu vstupného frontu:

- Most Occupied – dáta z najobsadenejšieho vstupného frontu sa dostávajú na výstup
- Round Robin – spravodlivý spôsob, ktorý vyberá z frontov obsahujúcich aspoň jeden kompletný paket. Ak takýto front neexistuje, použije sa spôsob Most Occupied
- Framed – spravodlivý spôsob, ktorý vyberá z frontov obsahujúcich aspoň jeden kompletný paket. Ak takýto front neexistuje, binder čaká.



Obrázok 3.2: FrameLink Binder

Fifo

Fifo predstavuje klasický front pre uchovávanie paketov. Okrem vstupného a výstupného FrameLinkového rozhrania disponuje navyše kontrolným, ktoré poskytuje informácie o stave frontu:

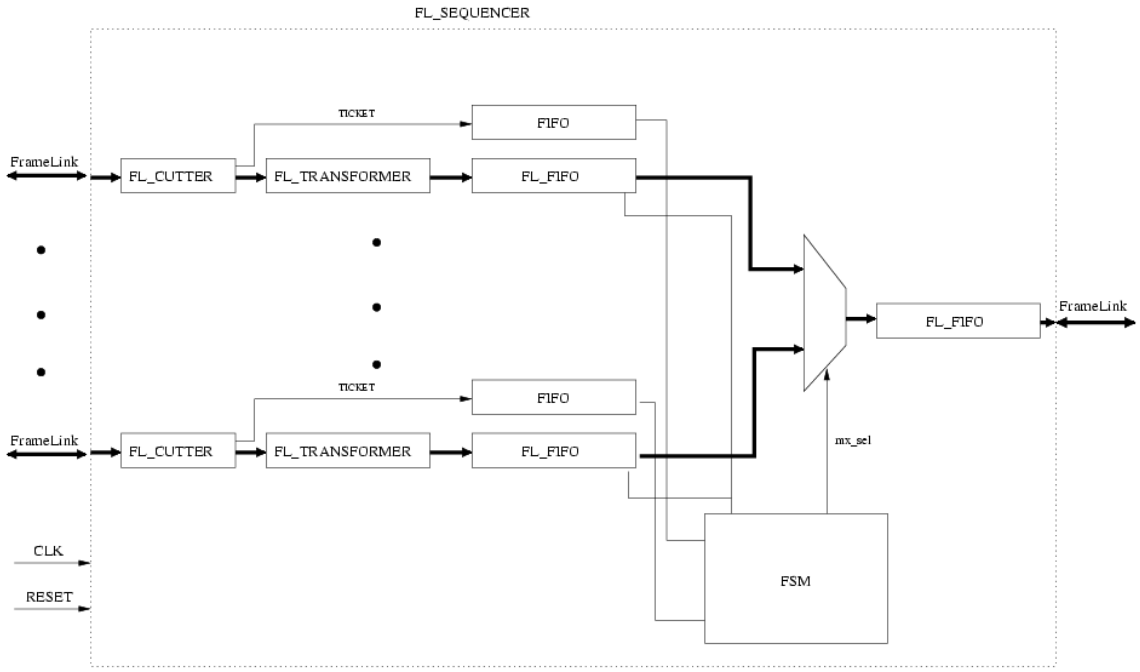
- EMPTY – front je prázdny.

- FULL – front je plný.
- FRAME_RDY – vo fronte sa nachádza aspoň jeden kompletný paket.
- LSTBLK – vo fronte je N alebo menej ako N volných položiek (N je generik).
- STATUS – zobrazuje niekoľko najvýznamnejších bitov čítača voľného miesta (počet bitov sa nastavuje generikom).

Kvôli zreťazenému spracovaniu sa môžu signály LSTBLK, STATUS, FULL a EMPTY líšiť od skutočnosti o jednu alebo dve položky.

Sequencer

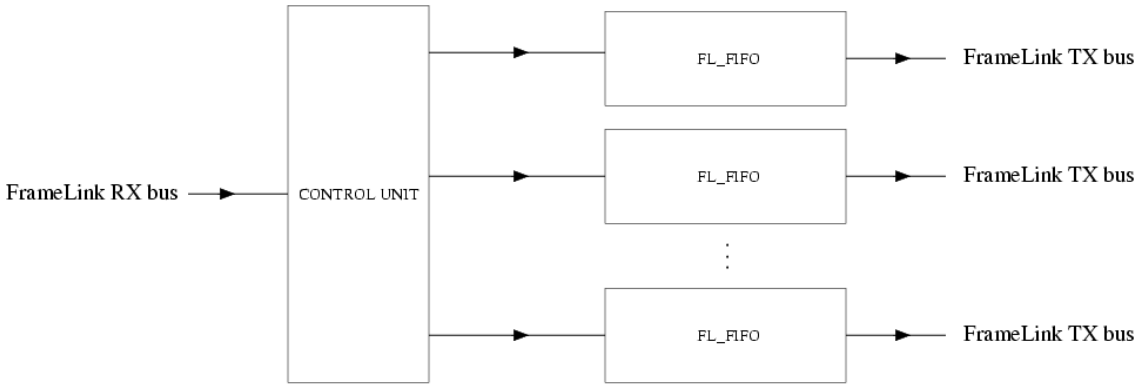
Sequencer plní podobnú funkciu ako Binder, a teda spája viacero vstupných tokov do jedného, pričom môžu mať ľubovoľnú dátovú šírku. Rozdiel je ale v tom, že každý paket si nesie v jednej zo svojich častí poradové číslo – tiket, ktoré určuje, v akom poradí sa pakety dostávajú na výstup. Sequencer zároveň tieto tikety z paketov odstraňuje.



Obrázok 3.3: FrameLink Sequencer

Splitter

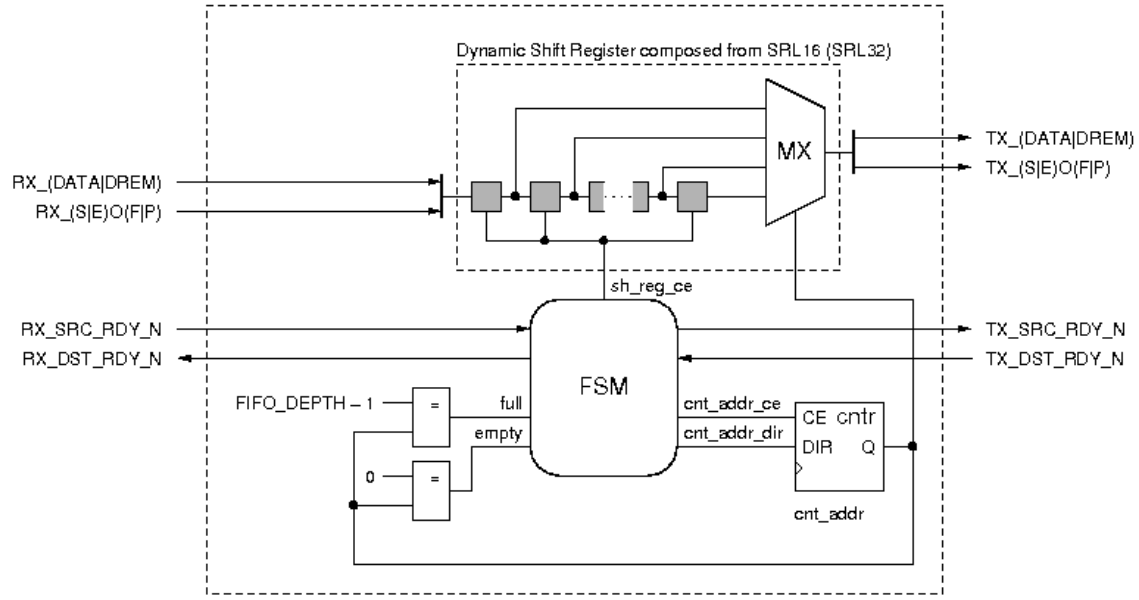
Splitter rozdeľuje jeden vstupný dátový tok na niekoľko výstupných. Plní teda presne opačnú funkciu ako Binder. Vstupný aj výstupný tok majú rovnakú šírku, výstupné rozhranie sa vyberá na základe najmenej zaplneného frontu.



Obrázok 3.4: FrameLink Splitter

Relay

Relay je malá FIFO pamäť s FrameLinkovým rozhraním vkladaná medzi FrameLinkové komponenty na prerušenie možných kritických ciest medzi komponentmi. FIFO pozostáva z posuvného registra SRL16 (SRL 32 pre Virtex-5) a je teda šetrné na zdroje. Možnosť registrovať všetky výstupné signály dovoľuje lepšie umiestnenie prepojených FrameLinkových komponentov na čipe.



Obrázok 3.5: FrameLink Relay

Kapitola 4

Verifikácia nástrojov pre protokol FrameLink

Tvorba hardvérového dizajnu pozostáva vo všeobecnosti z troch základných krokov. V prvej fáze špecifikujeme, akú funkcionalitu od komponentu očakávame a aké požiadavky naň budú v skutočnej prevádzke kladené. Potom prechádzame k návrhu architektúry dizajnu a nakoniec k samotnej implementácii. Podobnými krokmi by sa mal uberať aj proces verifikácie.

4.1 Verifikačné požiadavky

Táto prvá fáza vychádza priamo z hardvérovej špecifikácie a má za úlohu odpovedať na tri základné otázky:

- Čo má dizajn robiť?
- Čo nesmie dizajn robiť?
- Aká funkcionalita nie je pokrytá verifikačným plánom?

Čo má a čo nesmie dizajn robiť sa v našom prípade týka najmä korektnej práce so signálmi FrameLinkového rozhrania. Všetky nástroje, ktoré sú predmetom našej verifikácie, disponujú rovnakým rozhraním, a teda základné verifikačné požiadavky majú rovnaké:

1. Na dátovom porte sú platné dáta, ak je aktívne SRC_RDY_N a DST_RDY_N.
2. SOF_N je aktívny zároveň so SOP_N, ak je aktívny SRC_RDY_N.
3. EOF_N je aktívny zároveň s EOP_N, ak je aktívny SRC_RDY_N.
4. S aktívnym EOP_N, SRC_RDY_N a DST_RDY_N je platný signál REM.
5. Medzi EOF_N a SOF_N sa nevyskytujú dáta (SRC_RDY_N a DST_RDY_N nie sú aktívne súčasne).
6. Po každom SOF_N nasleduje (po určitom čase) signál EOF_N.
7. Po každom SOP_N nasleduje (po určitom čase) signál EOP_N.

8. Signál SOF_N nesmie byť aktívny po predchádzajúcom SOF_N, ak medzi nimi nebol aktívny EOF_N.

9. Signál SOP_N nesmie byť aktívny po predchádzajúcom SOP_N, ak medzi nimi nebol aktívny EOP_N.

Prípadné ďalšie požiadavky už musíme definovať pre konkrétne nástroje. Zároveň je nutné uviesť, akú funkcionalitu nebudeme v našej verifikácii overovať.

Splitter, Binder

Splitter a Binder sú nástroje s viacerými vstupnými/výstupnými rozhraniami. Disponujú teda určitými mechanizmami, na základe ktorých vyberajú konkrétne rozhranie. Správny výber rozhrania s ohľadom na zvolený mechanizmus však naša verifikácia nebude kontrolovať.

Sequencer

Pri Sequenceri je naopak správny výber vstupného rozhrania, daný číslom tiketu, ktorý si paket nesie, veľmi dôležitý, preto pridáme ďalšie tri požiadavky:

10. Čísla tiketov tvoria neprerušenú sekvenciu.

11. Pakety sa na výstup dostávajú v poradí určenom tiketmi.

12. Tiket je v priebehu spracovania z paketu odstránený.

Fifo

Fifo obsahuje navyše oproti ostatným komponentom ešte kontrolné rozhranie, pre ktoré je takisto nutné špecifikovať verifikačné požiadavky:

13. Ak je front prázdny, je aktívny signál EMPTY.

14. Ak je front plný, je aktívny signál FULL.

15. Ak sa vo fronte nachádza aspoň jeden kompletný paket, je aktívny signál FRAME_RDY.

16. Ak je vo fronte najviac N voľných položiek, je aktívny signál LSTBLK.

17. STATUS zobrazuje stav voľného miesta vo fronte (zvolený počet najvýznamnejších bitov).

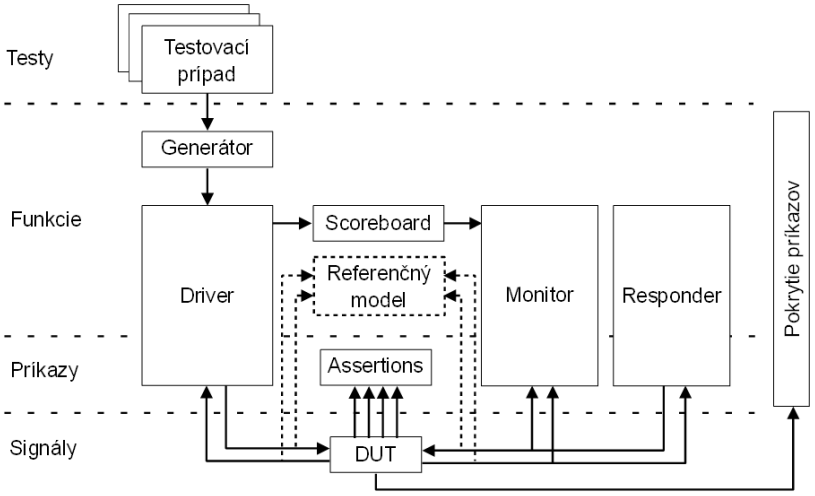
18. Pri signáloch EMPTY, FULL, LSTBLK a STATUS je povolená nepresnosť jedna až dve položky.

4.2 Návrh verifikačného prostredia

Druhá fáza sa zaoberá návrhom testbenchu. Základné úlohy verifikačného prostredia sú aplikácia stimulov na DUT, odchytávanie a kontrola správnosti odpovedí a meranie progresu. Návrh teda musíme vytvárať s ohľadom na čo najjednoduchšie plnenie týchto úloh.

Jednou z možností je využitie predpripravenej sady verifikačných komponentov ponúkaných niektorou z existujúcich metodológií, napríklad OVM alebo VMM [3]. Tento variant však nevyužijeme, pretože s ohľadom na nástroje, ktoré máme verifikovať, sú obe metodológie zbytočne robustné. Vytvoríme si množinu vlastných tried, ktoré však budú dodržiavať niektoré princípy metodológií, ako je constrained-random testovanie alebo coverage-driven testbench.

Teoretická štruktúra testbenchu sa skladá z piatich úrovní. V našom prostredí ale vynecháme typický generátor scenárov. Využijeme iba atomický generátor, ktorý bude produkovať jednotlivé transakcie, nie celé scenáre. Tvorbu scenára bude zabezpečovať funkčná vrstva. Návrh verifikačného prostredia je na obrázku 4.1. Jeho základné črty boli prevzaté z [5].



Obrázok 4.1: Verifikačné prostredie FrameLinkových nástrojov

Prenos FrameLinkovým rozhraním sa odohráva vo forme paketov. Paket pre nás predstavuje transakciu (stimul), ktorú budeme aplikovať na dizajn. Popis transakcií sa nachádza v triede *FrameLinkTransaction*. Počet častí paketu je definovaný v konkrétnom testovacom prípade, rovnako aj ich rozsahy. Samotný dátový obsah nie je z hľadiska funkčnosti FrameLinkových nástrojov dôležitý a preto sa náhodne generuje. Inštancie transakcií sú vytvárané pomocou generátora *Generator*. Ten zvolí dĺžku jednotlivých častí podľa stanoveného rozsahu a vygeneruje dáta. Stimuly sa prostredníctvom mailboxu predávajú driveru.

Vo verifikačnom prostredí sa nachádza iba jeden driver. Je popísaný triedou *FrameLinkDriver* a zastupuje funkcie ako proaktívneho tak aj reaktívneho drivera. Toto riešenie bolo zvolené s ohľadom na skutočnosť, že signál RX_SRC_RDY_N neslúži iba ako klasický signál oznamujúci pripravenosť komponentu, ale zároveň potvrdzuje aj platnosť dát na dátovom porte rozhrania. *FrameLinkDriver* vyberá z mailboxu transakcie, ktoré mu zašle generátor a aplikuje ich na rozhranie verifikovaného dizajnu. Každú transakciu následne posiela do *Scoreboard*u, kde prebehne jej porovnanie s odpoveďou dizajnu. Vkladanie ča-

kacích stavov nastavovaním signálu RX_SRC_RDY_N sa deje náhodne podľa parametrov zvolených v konkrétnom testovacom prípade.

O výstupné rozhranie DUT sa zaujímajú objekty dvoch tried. Trieda *FrameLinkResponder* predstavuje reaktívny monitor. Jeho úloha je rovnaká ako v prípade reaktívnej časti *FrameLinkDriver*u, a teda simulácia nepripravenosti komponentu prijímať dáta, tentokrát manipuláciou so signálom TX_DST_RDY_N. Druhým objektom je inštancia triedy *FrameLinkMonitor*. Jedná sa o klasický pasívny monitor. Prijíma pakety zasielané z výstupného rozhrania dizajnu a odosiela ich na porovnanie do *Scoreboard*u.

Scoreboard je jedným z dvoch samokontrolných mechanizmov, ktoré sa vyskytujú v základnom návrhu verifikačného prostredia. Dátovú štruktúru pre uloženie transakcií predstavuje transakčná tabuľka. Vyhľadávanie prebieha v závislosti na požiadavkách konkrétneho verifikovaného nástroja. V prípade, že nezáleží na poradí výstupu transakcií z dizajnu, sa prehľadáva sekvenčne celá tabuľka a porovnávajú sa jednotlivé pakety. Ak je poradie výstupu paketov striktne dané a treba ho dodržiavať, zvolíme variant, kedy sa s tabuľkou pracuje ako s klasickým FIFO a porovnáva sa iba jediný paket na začiatku frontu. Úlohu transformačnej funkcie preberá sada *callback* funkcií. Dve sú dostupné pre driver a dve pre monitor. V prípade drivera umožňujú modifikovať paket pred odoslaním do DUT a následným vložením do transakčnej tabuľky alebo iba pred samotným vložením do tabuľky. Podobne v monitore poskytuje možnosť predpripraviť objekt triedy *FrameLinkTransaction* na prijatie paketu alebo ho modifikovať pred odoslaním do scoreboardu na porovnanie. Tieto funkcie však využijeme iba pri verifikácii Sequenceru.

Druhým samokontrolným mechanizmom sú *assertions*. Zvolením vhodných sekvencií a vlastností jednoducho popíšeme náležitosti FrameLinkového rozhania a zabezpečíme tak dodržiavanie nízkoúrovňových pravidiel pre prenos.

Posledným komponentom spoločným pre verifikačné prostredie všetkých FrameLinkových nástrojov je *CommandCoverage*. Nejedná sa o klasickú jednotku sledujúcu funkčné pokrytie testovaného dizajnu. Verifikované nástroje poskytujú iba jednu konkrétnu funkcionalitu a tak plnohodnotný functional-coverage by v tomto prípade nenašiel uplatnenie. *CommandCoverage* je pripojený na vstupné a výstupné rozhranie DUT, kde sleduje, aké kombinácie signálov nastali. Progres verifikácie sa teda odvíja od vygenerovania všetkých prípustných kombinácií, na ktoré musí byť dizajn schopný zareagovať.

Verifikačné prostredie tak ako je popísané vyššie pokrýva potreby nástrojov s jedným vstupným a jedným výstupným rozhraním (Transformer). Nástroje s viacerými vstupnými/výstupnými rozhraniami vyžadujú našťastie iba minimálnu modifikáciu – pripojenie ďalších driverov/monitorov (responderov) tak, aby každé rozhranie disponovalo práve jedným.

Sequencer vyžaduje, aby pakety, ktoré mu prichádzajú na vstup, mali na konkrétnom mieste danom generikmi sekvenčné číslo – tiket. Dáta teda nemôžu byť úplne náhodné. Intuitívnym riešením by mohlo byť obmedzenie generátora, aby na požadovaných bitoch produkoval čísla tiketov. To by však neviedlo k úspechu, pretože každý driver pripojený k vstupnému rozhraniu má vlastný generátor a nepodarilo by sa nám zvyšovať sekvenčné číslo globálne bez ohľadu na to, na ktoré rozhranie transakcia prichádza. Riešenie sa skrýva v použití *callback* funkcií. V celom verifikačnom prostredí sa nachádza iba jeden *Scoreboard* a teda všetky drivery budú využívať spoločnú *callback* funkciu. Generátory vytvoria náhodnú transakciu, predajú ju driveru a ten pred jej aplikáciou na dizajn zavolá *callback*, ktorý doplní na patričné miesto tiket. Tak sa zaručí unikátnosť a sekvenčnosť tiketov v rámci celého prostredia.

Vzhľadom na to, že nástroj Fifo disponuje ešte jedným rozhraním, jeho testbench sa

bude od ostatných mierne líšiť. Správne nastavovanie kontrolných signálov overíme pripojením ďalšieho samokontrolného mechanizmu – referenčného modelu. Model, reprezentovaný triedou *FrameLinkFifoChecker*, vyhodnotí vstupy a výstupy DUT, na ich základe predikuje očakávané hodnoty kontrolného rozhrania a porovná so skutočnosťou. Dostatočné pokrytie jednotlivých kombinácií signálov, ktoré sa môžu na kontrolnom rozhraní vyskytnúť, budeme takisto sledovať pomocou *CommandCoverage*.

Overenie verifikačných požiadaviek

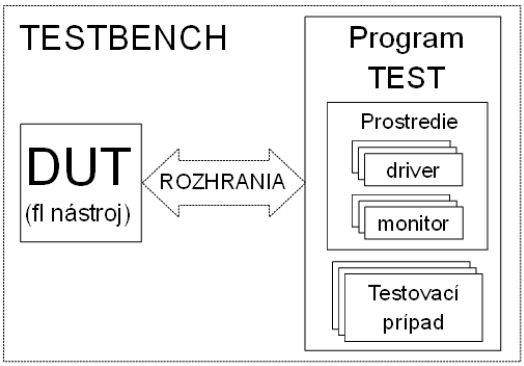
Návrh poskytuje celkovo tri samokontrolné mechanizmy, ktoré by mali zabezpečiť overenie všetkých stanovených verifikačných požiadaviek.

Požiadavky 1, 4, 5 a 12 sa týkajú správnosti dátového obsahu paketov. Ich overenie teda zabezpečíme porovnávaním očakávaných a získaných odpovedí v scoreboarde. Požiadavka 10 je splnená pridávaním tiketov v callback funkcii scoreboardu, ktorá je jednotná pre všetky drivery. Pri zabezpečení požiadavky 11 využijeme režim scoreboardu, ktorý umožňuje pracovať s transakčnou tabuľkou ako s FIFO frontom. Pakety do nej nie sú vkladané až po aplikovaní na DUT, ale v tej istej callback funkcii, ktorá pridáva do paketu tiket. To spôsobí, že sa v transakčnej tabuľke nachádzajú v takom poradí, v akom by mali prichádzať na výstup dizajnu. Porovnanie odpovede následne prebehne iba s paketom na začiatku frontu.

Požiadavky 2, 3, 6, 7, 8 a 9 sa týkajú závislostí medzi jednotlivými signálmi. Dajú sa efektívne popísať použitím temporálnych výrazov a preto na ich overenie použijeme assertions.

Korektnosť údajov poskytovaných kontrolným rozhraním nástroja Fifo (požiadavky 13, 14, 15, 16, 17, 18) je sledovaná referenčným modelom. V každom hodinovom cykle sa vyhodnocujú signály vstupného a výstupného rozhrania testovaného dizajnu, predikujú sa očakávané hodnoty kontrolného rozhrania a porovnávajú so skutočnými.

4.3 Implementácia verifikačného prostredia



Obrázok 4.2: Pripojenie prostredia a DUT

Implementácia prostredia pozostáva z troch základných častí – modul DUT, sada rozhraní, program TEST. V module DUT sa vytvára konkrétna inštancia verifikovaného dizajnu s požadovaným nastavením generikov a pripájajú sa jednotlivé porty k rozhraniu vytvorenému v jazyku SystemVerilog. Sada rozhraní obsahuje definície jednotlivých signálov

a ich zoskupenie pomocou *modportov*. Pre každý transaktor je vytvorený jeden modport, ktorý určuje, v akom smere signály používa (proaktívny a reaktívny transaktor má vstupné aj výstupné signály, pasívny iba vstupné). Na tomto mieste sú zároveň popísané aj assertions týkajúce sa vzťahov medzi signálmi. Keďže pri nástrojoch pracujúcich s protokolom FrameLink sa dá genericky nastaviť šírka dátového portu, rozhranie takisto disponuje generikom pre určenie šírky odpovedajúceho portu.

Tretia časť je program TEST. Má na starosti vykonanie samotnej verifikácie a pozostáva z volania jednotlivých testovacích prípadov. Testovacie prípady majú nasledovnú štruktúru:

```
task test();
  createEnvironment();          \*Vytvorenie objektov verifikačných komponentov*\
  enableEnvironment();          \*Povolenie činnosti verifikačných komponentov*\
  runGenerator();                          \*Spustenie generovania transakcií*\
  waitForGenerator();                   \*Čakanie na dokončenie generovania*\
  disableEnvironment();         \*Ukončenie činnosti verifikačných komponentov*\
  displayScoreboard();                     \*Zobrazenie stavu scoreboardu*\
  displayCoverage();                         \*Zobrazenie stavu pokrytia*\
endtask: test
```

Testovací prípad najprv vytvorí inštancie všetkých verifikačných komponentov a nastaví ich atribúty podľa zvolených parametrov. Rozličné nastavenie atribútov od seba odlišuje jednotlivé testovacie prípady. Zvyšné fázy sú už spoločné. Na konci každého testu sa vypisujú štatistiky v podobe zobrazenia scoreboardu a stavu pokrytia.

Doba testovania nie je časovo ohraničená, ale závisí na počte transakcií, ktoré chceme testovaným dizajnom preniesť. Požadovaný počet stimulov sa zadáva pri spúšťaní generátora. V cykle sa generujú náhodné transakcie a posielajú sa pomocou mailboxu do drivera. Po dosiahnutí požadovaného počtu sa generátor automaticky ukončí.

Trieda mailbox jazyka SystemVerilog umožňuje obmedziť dĺžku vnútorného FIFO frontu. V takom prípade je vkladanie do mailboxu blokujúce – ak v ňom nie je miesto pre ďalšiu položku, metóda vkladania čaká, kým sa neuvoľní. V našom prípade sme zmenšili front na jedinú položku a zabránili tak situácii, kedy sa transakcie vygenerovali príliš rýchlo, generátor sa ukončil a testovací prípad následne zastavil činnosť zvyšných verifikačných komponentov. Driver vtedy stihol odoslať iba minimum transakcií z tých, ktoré mal pripravené v mailboxe.

Podobne aj vyberanie položiek z prázdneho mailboxu je blokujúca činnosť, takže driver nie je potrebné vypnúť okamžite po odoslaní poslednej transakcie. Ak sa v mailboxe nachádza transakcia, driver ju vyzdvyhne a zavolá na ňu prvú callback funkciu, ktorá umožňuje modifikovať stimuly ešte pred odoslaním do DUT. Následne ju predá metóde, ktorá má na starosti jej aplikovanie na dizajn. Pred a behom odosielania sa podľa zvolených parametrov náhodne vkladajú čakacie stavy. Volanie druhej callback funkcie umožňuje ešte dodatočnú modifikáciu a zabezpečuje umiestnenie transakcie do scoreboadu.

Monitor poskytuje presne opačnú funkcionalitu ako driver, ale ich základné kostry sa prakticky nelíšia. Po zavolaní prvej callback funkcie (v našom verifikačnom projekte nenašla uplatnenie) sa činnosť predáva metóde, ktorá je schopná zo sledu signálov prichádzajúcich z DUT zostaviť transakciu. Transakcia sa následne pomocou druhej callback funkcie, ktorá rovnako poskytuje možnosť dodatočnej modifikácie, predloží porovnávacej funkcii. Ak porovnávacia funkcia nájde odpovedajúcu transakciu v transakčnej tabuľke, odstráni ju. Ak ale k zhode nedôjde, callback vypíše chybovú hlášku a ukončí testovanie.

Oddelením reaktívneho a pasívneho monitora vznikla trieda responder. Jej jedinou úlohou je vkladanie čakacích stavov. Jedná sa o náhodný proces, ktorý ovplyvňujeme nastavovaním jednotlivých atribútov respondera. Responder, podobne ako driver, ponúka osobitnú kontrolu nad vkladaním oneskorení medzi transakcie a počas prijímania transakcie.

Samokontrolný mechanizmus scoreboarding pozostáva z troch častí – transformačná funkcia, dátová štruktúra, porovnávacia funkcia. Trieda Scoreboard v skutočnosti vytvára iba obálku nad týmto mechanizmom. Transformačné funkcie poskytujú callback triedy a porovnávacia funkcia je súčasťou definície transakcie. Dátová štruktúra je implementovaná triedou TransactionTable. Umožňuje dvojaké vyhľadávanie v transakčnej tabuľke – prehľadanie celej tabuľky alebo striktný FIFO prístup – a súbežným prístupom sa vyhýba pomocou zamykania semaforom.

Druhým samokontrolným mechanizmom, využitým pri verifikácii Fifo, je referenčný model. Na základe stavov signálov na FrameLinkových rozhraniach vypočíta hodnoty LSTBLK, STATUS, EMPTY, FULL a FRAME_RDY a očakáva, že o takt neskôr obdrží rovnaké hodnoty prostredníctvom kontrolného rozhrania od DUT.

Posledným komponentom, ktorý sa nachádza v našom verifikačnom prostredí, je trieda CommandCoverage. Zaznamenáva pokrytie rôznych hodnôt signálov na FrameLinkových rozhraniach. Základom je vytvorenie *covergroup* a v rámci nej definovanie jednotlivých bodov pokrytia. Nás zaujímajú predovšetkým kombinácie riadiacich signálov a hodnoty signálu REM. Samozrejme nie všetky kombinácie signálov sú legálne (nastavenie SOF_N bez SOP_N, ...), prípadne na niektoré nie sme zvedaví (hodnota REM ak je EOP_N neaktívne), preto ich zo štatistík vylúčime. Po spustení trieda už len v každom takte zavolá vzorkovaciu funkciu.

```
covergroup CommandsCovergroup;
  sof:  coverpoint SOF_N;                                  \*Pokrytie signálu SOF_N*\
  eof:  coverpoint EOF_N;                                  \*Pokrytie signálu EOF_N*\
          ⋮
  rem:  coverpoint REM;                                    \*Pokrytie signálu REM*\

  cross sof, sop, eof, eop, src_rdy, dst_rdy {   \*Pokrytie kombinácií signálov*\
    illegal_bins il_val0 = binsof(sop.1) && binsof(sof.0);        \*Nelegálne*\
    illegal_bins il_val1 = binsof(eop.1) && binsof(eof.0);        \*kombinácie*\
  }
  cross eop, rem {
    ignore_bins ig_val = binsof(eop.1);                \*Nezaujímavé kombinácie*\
  }
endgroup: CommandsCovergroup

task run();
  while (enabled) begin
    @(fl.fl_cb);                                       \*Prechod na ďalší hodinový takt*\
    CommandsCovergroup.sample();          \*Vzorkovanie aktuálnych hodnôt signálov*\
  end
endtask: run
```

Pokrytie sa sleduje na každom rozhraní zvlášť. Verifikačné prostredie nástroja Fifo vykonáva podobné štatistiky aj nad kontrolným rozhraním.

4.4 Výsledky verifikácie

Spôsob reportovania chyby vo verifikovanom dizajne závisí od typu samokontrolného mechanizmu, ktorý ju odhalil. Assertions oznamujú chyby prostredníctvom výpisu hlášky. Za hodnotou simulačného času, v ktorom chyba nastala, nasleduje textová správa definovaná na mieste implementácie assertionu. Nedodržanie assertions je takisto dobre vidieť v časovom priebehu signálov (waveform), kde ModelSim daný čas označí červenou šípkou. Porušenie assertions nespôsobí ukončenie testovania.

Odhalenie chyby pomocou scoreboardingu sa prejavuje iným spôsobom. Ak scoreboard obdrží z monitora transakciu, ktorá sa nezhoduje so žiadnou resp. prvou v transakčnej tabuľke, vypíše ju na obrazovku a ukončí testovanie. Zároveň zobrazí aj aktuálny obsah transakčnej tabuľky. Okamžité ukončenie verifikácie pomáha ľahšie dohľadať miesto, v ktorom k chybe došlo. Rovnako po dokončení každého testu musí byť obsah transakčnej tabuľky prázdny, opačný stav znamená, že sa nám nejaké transakcie v dizajne stratili.

Referenčný model použitý vo verifikácii FrameLinkového nástroja Fifo sa správa pri oznamovaní chýb rovnako ako assertions. Ani tu výpis hlášky a označenie v časovom priebehu signálov nespôsobia prerušenie testovania.

Existujú však aj chyby, ktoré žiaden z použitých mechanizmov nereportuje, a odhalí ich iba verifikátor pohľadom do simulácie. Najčastejším prípadom je zaseknutie sa testovaného dizajnu v situácii, kedy vloží čakací stav. Testbench čaká, kým bude DUT opäť schopná činnosti, ale k tomu už nedôjde a simulácia nikdy neskončí. Rozoznať, či sa dizajn zasekol a verifikácia už nevykonáva žiadnu užitočnú činnosť, zostáva na verifikátorovi.

Správna činnosť dizajnu je teda overená v prípade, že

- verifikácia sa korektne ukončí,
- počas testovania sa nevyskytnú žiadne chybové hlášky,
- transakčná tabuľka je na konci testu prázdna,
- pokrytie sa blíži hodnote 100%.

Transformer

Transformer disponuje dvoma generickými parametrami, ktoré nastavujú dátovú šírku vstupného a výstupného rozhrania. Môžu byť nastavené na 8, 16, 32, 64 alebo 128 bitov, čo nám spolu dáva 25 kombinácií, ktoré bolo treba zverifikovať. Aby sme dosiahli pokrytie 100%, bolo treba vytvoriť viacero testovacích prípadov. Počet častí v pakete sa pohyboval v rozmedzí 1 až 3, dĺžka jednotlivých častí bola 1 – 1536 bajtov a testovacie prípady sa líšili aj vkladaním čakacích stavov. Niekedy sa negenerovali žiadne a dizajn tak bežal na najvyššej rýchlosti, inokedy sa vkladali dosť intenzívne. Chyby pri tomto nástroji neboli zistené.

Fifo

Fifo poskytuje vyšší počet generikov. Okrem nastavenia dátovej šírky, ktorá je tentokrát jednotná pre vstupné aj výstupné rozhranie a podporuje rovnaké hodnoty ako pri Transformeri, umožňuje zadať počet položiek, ktoré sa do neho zmestia, veľkosť bloku (ovplyvňuje nastavovanie LSTBLK), šírku kontrolného signálu STATUS a nastavenie použitia *Selectable BRAM* alebo *Distmem FIFO*. Rovnako je nutné generikom určiť, koľko častí bude paket

obsahovať. To spôsobí, že pokrytie pre konkrétne nastavenie parametrov bude vždy menej ako stopercentné (pri počte častí 2 a 3 sa nám nepodarí naraz nastaviť SOF_N, SOP_N, EOF_N a EOP_N a naopak). Testovanie všetkých kombinácií generikov by bolo časovo náročné. Overia sa preto len tie, ktoré sa využívajú v zapojení v skutočnom prostredí, alebo sa vyberú náhodne. Pri použití *BRAM* vnútorné zreťazenie spôsobuje, že dáta sa na výstup dostanú až o tak alebo dva neskôr a signály LSTBLK, STATUS, EMPTY a FULL môžu mať odchýlku jedna až dve položky. Ani v tejto verifikácii sa chyby nenašli.

Binder

Genericky sa dá pri Binderi nastaviť vstupná dátová šírka, počet vstupných rozhraní, výstupná dátová šírka (nesmie byť väčšia ako násobok predchádzajúcich dvoch genrikov), počet častí paketu a spôsob výberu vstupného frontu. Pri využití spôsobu FRAMED, kedy sa vyberá z frontov obsahujúcich aspoň jeden kompletný paket, dochádzalo k strate transakcií. Po dokončení testovacieho prípadu sa nezhodoval počet pridaných a odobratých transakcií v transakčnej tabuľke (A.1).

Binder nakoniec prešiel kompletnou prestavbou, bol vybudovaný nad komponentom NFIFO2FIFO a umožňuje využiť LUT miesto BlockRAM pamätí. Existujúce verifikačné prostredie sa používalo na priebežné ladenie, takže po dokončení implementácie bol nástroj zároveň aj zverifikovaný.

Splitter

Okrem nastavenia vstupnej dátovej šírky a počtu výstupných rozhraní je opäť prítomný aj generik na určenie počtu častí paketu. Verifikácie prebehli na všetkých 48 možných kombináciách (dátová šírka 16/32/64/128 bitov, počet rozhraní 2/4/8/16, počet časí paketu 1/2/3) a neobjavila sa žiadna chyba. Avšak to, či Splitter naozaj vyberal najmenej zaplnený výstupný front verifikácie nedokážu overiť.

Sequencer

Aj keď Sequencer ponúka na prvý pohľad veľké množstvo generikov, v súčasnej implementácii fungujú iba niektoré kombinácie. Obmedzenia sa týkajú vstupnej dátovej šírky, ktorá je podporovaná iba v rozmeroch 16 a 128 bitov, a najmä umiestnenia a veľkosti tiketu. Musí sa nachádzať v prvej časti paketu na offsete 0 alebo 2 a dĺžka je presne dva bajty.

Verifikácia odhalila dve chyby. Prvá sa týkala zlej práce so signálmi RX_DST_RDY_N. Komponent sa zasekol pri vložení čakacieho stavu a k nastaveniu signálu späť do "0üž nedošlo (A.2). Verifikácia tým pádom neskončila, prostredie čakalo, kým bude Sequencer opäť schopný prijímať dáta. Chybu spôsobovala FIFO pamäť na tikety.

V druhom prípade dochádzalo k chybnému nastavovaniu TX_REM signálu, ktorý určuje počet platných bajtov v posledom slove danej časti paketu. Monitor teda prijal nesprávne dáta a po porovnaní v scoreboarde sa vypísala chybová hláška a verifikácia ukončila (A.3). Problém zapríčiňoval podkomponent Cutter_fake, konkrétne register, ktorý pracoval s hodnotou REM.

Po opravení uvedených chýb verifikácie už ďalšie nedostatky neodhalili.

Relay

Štyri generiky nástroja Relay umožňujú nastaviť dátovú šírku (jednotnú pre vstupné aj výstupné rozhranie), veľkosť vnútornej FIFO pamäte a pridať vstupný lebo výstupný register. O chybnom správaní tohto komponentu sa vedelo už predom a verifikácie to iba potvrdili. Po pridaní vstupného registru dochádza k situáciám, kedy Relay vystaví dvakrát po sebe tie isté dáta bez toho, aby ich zneplatnil signálom TX_SRC_RDY_N (A.4).

Medzičasom vznikol na projekte nástroj Pipe, ktorý plní prakticky rovnakú úlohu ako Relay. Keďže pri Pipe sa podobné chyby nevyskytovali a verifikácia zbehla bez problémov, Relay bol odstránený a nahradený týmto novým nástrojom.

Kapitola 5

Záver

Cieľom tejto práce bol návrh a implementácia verifikačného prostredia pre nástroje pracujúce s protokol FrameLink, ktoré sa využívajú na projekte Liberouter [7]. Nezanedbateľnú súčasť tvorilo takisto samotné testovanie dizajnov.

Na počiatku riešenia som sa zoznámil s jazykom SystemVerilog [10] [4], ktorý v súčasnosti predstavuje najrozšírenejšie používaný prostriedok pre verifikáciu hardvérových dizajnov, so zásadami a postupmi využívanými pri tvorbe testbenchov [6] [2] a základnými princípmi, ktoré ponúkajú dve svetovo najrozšírenejšie metodológie [9] [3]. Tento teoretický základ je zhrnutý v kapitole 2. Následne bolo potrebné oboznámiť sa s protokolom FrameLink, jeho rozhraním a štruktúrou prenášaných paketov, a jednotlivými nástrojmi, ktoré s uvedeným protokolom pracujú (kapitola 3).

Návrh a výsledná implementácia dobre zohľadňujú špecifiká FrameLinkových nástrojov, ktoré sú reprezentované najmä možnosťou genericky nastaviť dátovú šírku a počet vstupných a výstupných rozhraní. Výsledky verifikácie zhrnuté v kapitole 4 ukazujú, že testbench pokrýva nielen najbežnejšie prípady, v ktorých sa môže dizajn v skutočnej prevádzke vyskytnúť, ale aj mnohé okrajové situácie a zaručuje tak, že zverifikovaný dizajn obstojí v akýchkoľvek prípustných podmienkach.

Implementované riešenie je vhodné na verifikáciu ľubovoľného nástroja pracujúceho nad rozhraním FrameLink. Modularita prostredia umožňuje nasadiť jednotlivé verifikačné komponenty samostatne v iných testbenchoch a uľahčiť tak verifikáciu dizajnov, ktoré majú iba vstupné alebo výstupné FrameLinkové rozhranie.

Hoci verifikačné prostredie podporuje pri aplikácii stimulov vkladanie čakacích stavov, nedisponuje klasickým generátorom scenárov a neposkytuje teda možnosť vytvárať sofistikovanejšie scenáre, ktoré by mohli byť potrebné pri testovaní komplexnejších dizajnov. Druhé rozšírenie do budúcnosti by som videl v implementácii samokontrolného mechanizmu, ktorý by overoval správnosť prideľovania prostriedkov podľa zvolenej stratégie (výber frontu v Binderi a Splitteri).

Literatúra

[1] Anderson, T. L.: Open Verification Methodology: Fulfilling the Promise of SystemVerilog. *Information Quarterly*, ročník 7, č. 1, 2008: s. 52–54.

[2] Bergeron, J.: *Writing testbenches using SystemVerilog*. Springer, 2006, ISBN 0-387-31275-7.

[3] Bergeron, J.; Cerny, E.; Nightingale, A.; aj.: *Verification Methodology Manual for SystemVerilog*. Springer, 2006, ISBN 978-0-387-25538-5.

[4] IEEE Computer Society: *IEEE Std 1800-2005: IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*. 2005.

[5] Kobiersky, P.; Málek, T.; Puš, V.: SystemVerilog verification of VHDL design. Technická správa v1.0, Honeywell, august 2007.

[6] Spear, C.: *SystemVerilog for verification*. Springer, 2006, ISBN 0-387-27036-1.

[7] WWW stránky: Liberouter. http://www.liberouter.org.

[8] WWW stránky: LocalLink. http://www.xilinx.com/products/ipcenter/LocalLink_UserInterface.htm.

[9] WWW stránky: Open Verification Methodology. http://www.ovmworld.org.

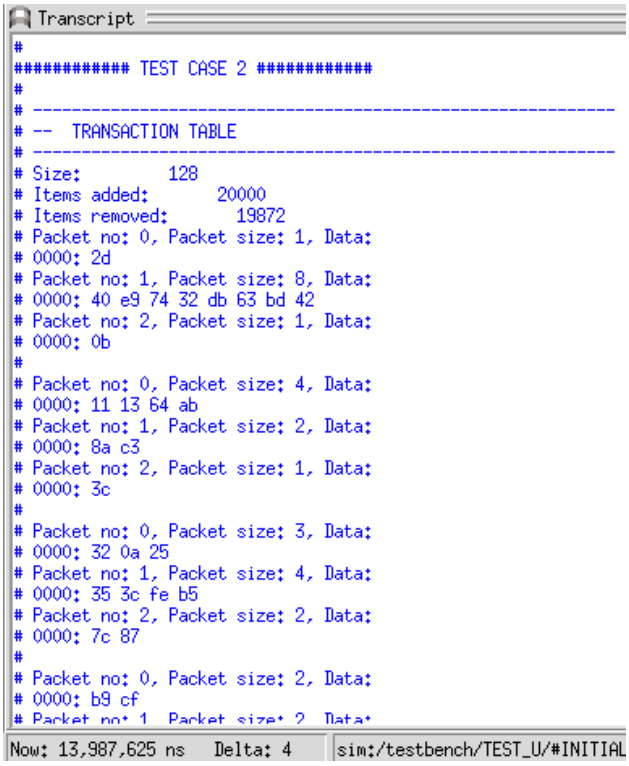
[10] WWW stránky: SystemVerilog. http://www.systemverilog.org.

[11] WWW stránky: Wikipedia — SystemVerilog. http://en.wikipedia.org/wiki/SystemVerilog.

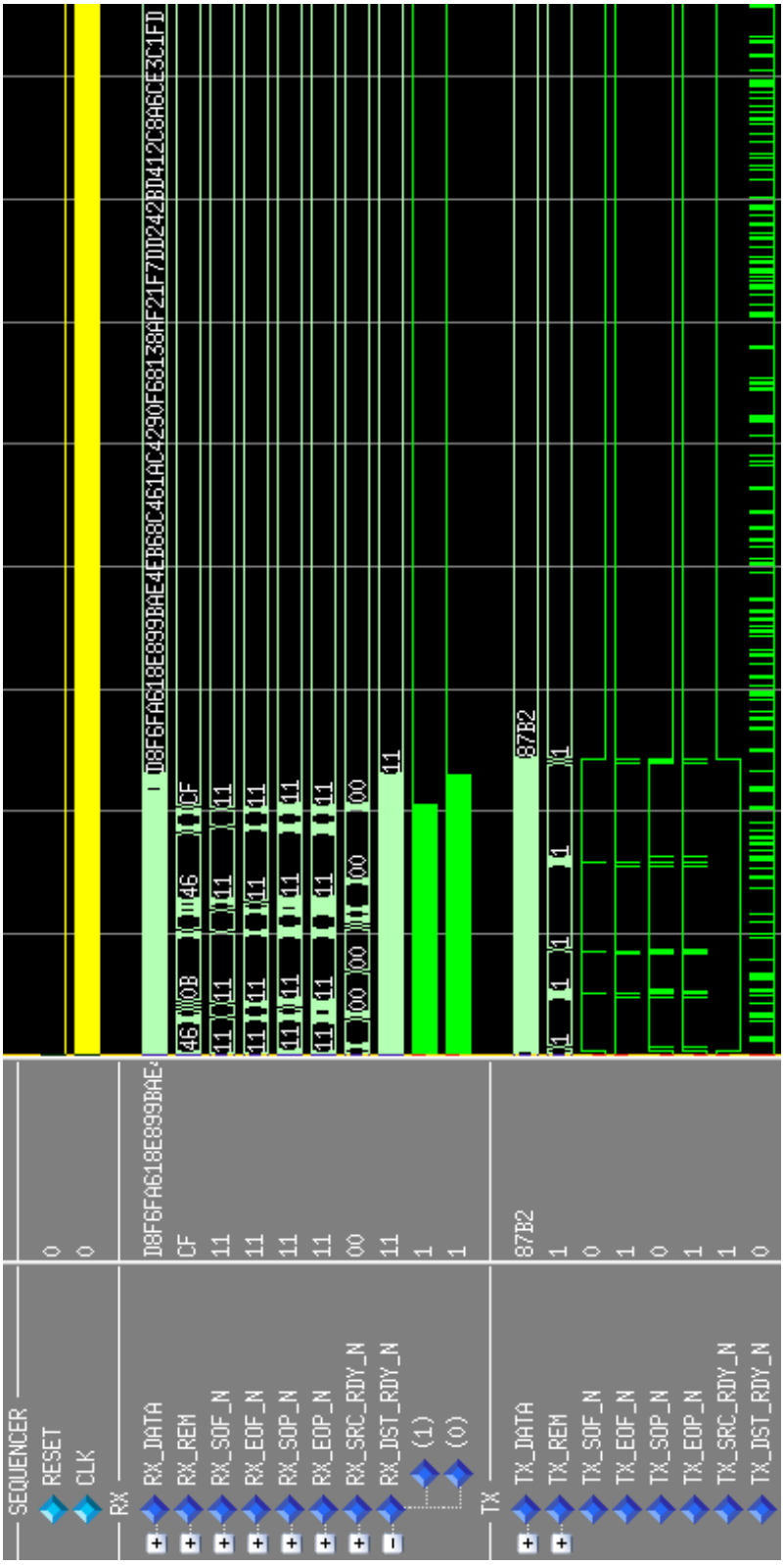
Dodatok A

Výsledky verifikácie – obrazová príloha

Príloha obsahuje obrázky zachytávajúce odhalené chybové stavy jednotlivých testovaných nástrojov.



Obrázok A.1: FrameLink Binder – transakčná tabuľka po skončení testu



Obrázok A.2: FrameLink Sequencer – zaseknutie dizajnu pri vložení čakacieho stavu

Obrázok A.3: FrameLink Sequencer – chybné nastavenie REM signálu

Obrázok A.4: FrameLink Relay – vystavenie rovnakých dát

Dodatok B

Obsah CD

K práci je priložené CD, ktoré obsahuje elektronickú verziu tejto technickej správy, jej zdrojové súbory a takisto aj zdrojové súbory verifikačného prostredia pre testované nástroje.