VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROSTŘEDÍ PRO VERIFIKACI DMA ŘADIČŮ V JAZYKU SYSTEMVERILOG

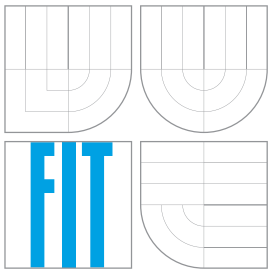
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARCELA ŠIMKOVÁ

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

PROSTŘEDÍ PRO VERIFIKACI DMA ŘADIČŮ V JAZYKU SYSTEMVERILOG

SYSTEMVERILOG FRAMEWORK FOR DMA CONTROLLERS VERIFICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARCELA ŠIMKOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. VIKTOR PUŠ
SUPERVISOR

BRNO 2009

Abstrakt

V dnešních hardwarových návrzích se verifikační techniky používají pro ověřování funkcionality dílčích komponent i komplexních systémů. Tato bakalářská práce se zabývá verifikací DMA řadičů. Jsou popsány teoretické principy verifikace v jazyce SystemVerilog a činnost DMA – přenos dat přes sběrnici bez účasti procesoru. Následuje úvod do praktické části verifikace řadičů, těžištěm práce je návrh verifikačního prostředí a následně samotná verifikace a její výsledky.

Abstract

In contemporary hardware design, verification techniques are exploited to verify the function of hardware components as well as complex systems. This thesis deals with functional verification of DMA controllers. It describes the theoretical principles of verification using the SystemVerilog language and the principles of DMA data transfer. The design of controllers is described, with the focus on design of the verification environment and results of the verification.

Klíčová slova

Verifikace, DMA, SystemVerilog.

Keywords

Verification, DMA, SystemVerilog.

Citace

Marcela Šimková: Prostředí pro verifikaci DMA řadičů
v jazyku SystemVerilog, bakalářská práce, Brno, FIT VUT v Brně, 2009

Prostředí pro verifikaci DMA řadičů v jazyku SystemVerilog

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Viktora Puše. Další informace mi poskytli členové týmu Liberouter. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Marcela Šimková
17. mája 2009

Poděkování

Především bych ráda poděkovala vedoucímu své bakalářské práce Ing. Viktoru Pušovi za odborné vedení a čas věnovaný konzultacím této práce. Dále bych chtěla poděkovat kolegům z projektu Liberouter za poskytnutí potřebných informací při návrhu verifikace a ladění nalezených chyb.

Obsah

Kapitola 1

Úvod

Verifikácia je dôležitou súčasťou tvorby hardwarových návrhov. Z tohto pohľadu má dva rôzne významy. Prvým je overenie ekvivalencie dvoch modelov návrhu. Kontroluje dve reprezentácie rovnakého návrhu, u ktorých by mali rovnaké podnety vyvolať rovnaké reakcie. Druhým významom je tzv. „property checking“. Zisťuje, či návrh spĺňa funkčné požiadavky a špecifikáciu.

V každom návrhu sa môžu objaviť chyby. Niektoré je možné identifikovať a lokalizovať jednoducho, prejavia sa viditeľným spôsobom. Iné nastanú len v hraničných situáciách (napr. zaplnenie vstupno-výstupných bufferov, prechod medzi stránkami v pamäti), či pri chybnej kombinácii signálov na rozhraniach. Pri ich odhaľovaní sa používajú rôzne postupy a prostriedky. Návrhár, ktorý implementuje časť systému alebo systém ako celok, overuje funkčnosť do takej miery, aby splnil základné požiadavky kladené na systém. Testuje jeho kľúčové funkcie. Skupina testerov overuje komplexnú funkcionalitu systému vrátane hraničných situácií. Hardwarové testy sú však často zdĺhavé. Podľa obtiažnosti systému môže testovanie trvať desiatky minút, hodín, tých zložitejších aj niekoľko dní. Preto je dôležitou súčasťou testovania verifikácia. V odbornej literatúre je popísané veľké množstvo spôsobov verifikácie hardwarových návrhov. Výber závisí od charakteru systému a od jeho komplexnosti. V projekte Liberouter som sa stretla s dvoma spôsobmi. Používame verifikáciu čisto formálnu, pracujúcu s matematickými výrokmi, formulami, vzťahmi a verifikáciu softwarovú v jazyku SystemVerilog.

SystemVerilog je verifikačný prostriedok, ktorý slúži na odhalenie rôznorodých chýb a nedostatkov. Ide napr. o chyby v časovaní, v nastavení signálov na rozhraniach, či v nesprávnom pochopení špecifikácie. Používa sa najmä na ladenie programov v jazykoch pre popis hardwaru ako je VHDL a Verilog. V projekte Liberouter našiel využitie pri ladení designov vo VHDL. Objavenie chýb verifikáciou uľahčuje prácu návrhárom hardwarového designu, pretože simulácia v SystemVerilogu trvá iba zopár minút. Preto je zjavne výhodnejšie objaviť chyby už v tomto štádiu. Šetríme nielen čas, ale aj prostriedky.

V praxi používame simulačný nástroj ModelSim firmy Mentor Graphics, ktorý podporuje SystemVerilog. Zobrazuje simuláciu vytvoreného návrhu, umožňuje sledovať pokrytie kódu (code coverage), rýchlo lokalizovať chyby, odhaliť a eliminovať mŕtve časti kódu.

Cieľom tejto bakalárskej práce je verifikácia DMA radičov. Princípu fungovania DMA prenosu sa venuje samostatná kapitola.

Pri návrhu verifikačného prostredia (testbench) som sa pridŕžala koncepcie navrhnutej v našom projekte [6] ako aj v príručke SystemVerilog for Verification od Chrisa Speara [9]. Zmienené budú aj nové techniky a postupy pri verifikácii v SystemVerilogu podľa metodológie OVM – Open Verification Methodology [2].

Kapitola 2

Princípy verifikácie v SystemVerilog

2.1 História vývoja verifikačných prostriedkov

S nárastom počtu nových technológií a postupov v hardwarovom návrhu za posledných 35 rokov rástla tiež potreba nájsť nové a efektívne spôsoby verifikácie (historický vývoj podľa [3]).

V 70-tych rokoch minulého storočia sa verifikovalo najmä formálne, prostredníctvom detailných návrhov na papieri, matematickej logiky a dôkazov. Počas 80-tych rokov sa stali populárnymi simulačné nástroje, ale prevládali proprietárne riešenia s rôznymi simulačnými prostrediami. Verifikácia tak do značnej miery závisela od údajov zo simulácie, ktoré bolo dané prostredie schopné poskytnúť. V roku 1987 sa štandardom hardwarového návrhu v Európe stal jazyk VHDL a všeobecne boli jazyky rodiny HDL (Hardware Description Languages) uznané ako primárny návrhový a verifikačný prostriedok.

V 90-tych rokoch sa objavili statické verifikačné nástroje, ktoré pomáhali pri statickej analýze hardwarových návrhov a riešení kompatibility s návrhovými pravidlami. Postupne sa začali používať inteligentné verifikačné prostredia v HDL, boli predstavené nové nástroje, techniky a postupy pri simulácii a verifikácii, ktoré zvyšovali ich popularitu.

S narastajúcou komplexnosťou návrhov bola verifikácia HDL jazykmi čoraz ťažšie uskutočniteľná. Situáciu vyriešil nástup jazykov rodiny HVL (Hardware Verification Languages). Využívajú princíp funkčného pokrytia, ABV (Assertion-based Verification) a umožňujú sledovať pokrok vo verifikačnom procese. Všetky zmienené techniky budú dopodrobna objasnené v nasledujúcom texte. Príkladom HVL jazykov sú: Property Specification Language (PSL), OpenVera Assertion Language (OVA), SystemVerilog (SV), Open Verification Library (OVL).

2.2 SystemVerilog

Evolúcia verifikačnej metodológie a jazykov viedla k vzniku štandardu SystemVerilogu (IEEE 1800–2005). Ide o komplexný programovací jazyk pre popis hardwaru, verifikáciu a podporu simulácie, obsahuje črty viacerých programovacích jazykov.

Dôležité znaky jazyka:

- *objektovo-orientovaný prístup v programovaní* – tvorba verifikačného prostredia na rôznej úrovni abstrakcie, znovupoužiteľnosť, udržovateľnosť a prehľadnosť kódu (jednoduchá dedičnosť, polymorfizmus), ochrana citlivých dát;
- *Constrained Random Testing* – podpora automatického generovania transakcií, ktorých formát je daný obmedzujúcimi podmienkami (constrains);
- *pokrytie (Coverage)* – zbieranie štatistík o dosiahnutí povolených kombinácií signálov na rozhraniach, pokrytí kódu a o funkčnom pokrytí návrhu;
- *rozhrania a assertions* – zoskupenie súvisiacich signálov do rozhraní, kontrola dodržiavania protokolov rozhraní;
- *spolupráca s inými programovacími jazykmi* – možnosť volať C/C++ funkcie zo SV (balíčky fukncií), vytvárať verifikačné prostredie s C++ rozhraním či riadiť verifikáciu z C++;

2.3 OVM – Open Verification Methodology

Efektívna verifikácia v SystemVerilogu vyžaduje metodológiu, ktorá špecifikuje, ako konštruovať znovupoužiteľné a spoločne pracujúce verifikačné prostredia (testbenche).

Metodológia OVM bola zverejnená 9. januára 2007 spoločnosťami Cadence Design Systems a Mentor Graphics ako spoločná snaha o vytvorenie metodológie pre SV verifikáciu. Poskytuje voľne dostupné knižnice, dokumentáciu a príklady.

Moderné verifikačné projekty používajú Constrained Random Testing a funkčné pokrytie stanovuje postup. Verifikácia je kompletná, iba keď sú overené všetky kľúčové funkcie návrhu. OVM podporuje tento postup ako najlepší spôsob verifikácie komplexného návrhu.

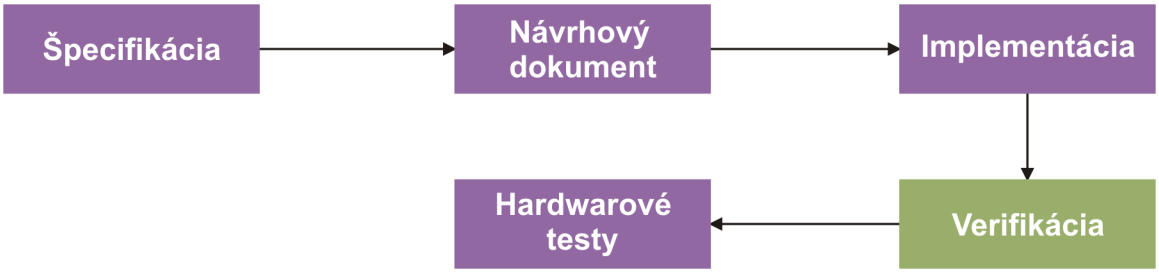
OVL – Open Verification Library je knižnica tried obsahujúca zdrojové kódy a dokumentáciu, je napísaná použitím štandardu SystemVerilogu IEEE 1800. Používa sa ako základ tvorby testbenchových architektúr i generátorov náhodných transakcií. Dokumentácia poskytuje návody, ako testbenche konštruovať a znovupoužívať jednotlivé komponenty [2].

2.4 Verifikačný proces

Cieľom hardwarového návrhu je vytvorenie zariadenia, ktoré spĺňa určitú úlohu. Účelom verifikácie tohto návrhu je overenie, že zariadenie túto úlohu vykonáva správne (je reprezentáciou špecifikácie). Chyby sa objavujú, keď zariadenie tieto podmienky nespĺňa.

Pri verifikácii bloku návrhu verifikátor potrebuje rozumieť formátu vstupu a výstupu (vstupné a výstupné signály, ich význam, špecifikácia prechodov medzi logickými stavmi), ako aj transformačnej funkcii, ktorá je implementovaná vo vnútri tohto bloku. Nepotrebuje však do detailov poznať jej implementáciu.

Odporúča sa, aby verifikáciu vykonávala iná osoba z projektového tímu, nie samotný návrhár, ktorý blok implementoval. Vytvoria sa tak nezávislé testy. Verifikátor môže pri ladení navrhnúť úplne iné metódy a postupy, ktoré návrhára nemuseli napadnúť. Ako už bolo zmienené v úvode, je výhodnejšie a časovo úspornejšie vykonávať verifikáciu ešte pred testovaním designu v hardwari. Správny postup vidno na obrázku 2.1.



Obr. 2.1: Postup pri tvorbe a ladení programu. Hardwarovým testom by mala predchádzať verifikácia.

Okrem hľadania chýb je dôležité testovať zariadenie tiež reverzným spôsobom. Zámerne sa snažíme produkovať chyby a sledujeme reakcie. Zariadenie by sa malo s takýmito situáciami korektne vysporiadať napr. zahodením dát, ktoré nespĺňajú špecifikáciu, generovaním varovných hlásení a pod.

2.5 Odhaľovanie chýb

Pri overovaní funkcionality systému sa najčastejšie postupuje od verifikácie najjednoduchších komponent smerom k väčším celkom, ktoré sa z týchto komponent skladajú. V súlade s týmto postupom vzrastá úroveň abstrakcie a objavujú sa rôzne typy chýb, preto testy vykonávame na rôznych úrovniach:

1. **Bloková úroveň (jadro komponenty)** – overujú sa funkcie jednoduchej komponenty (transformácia vstupov na výstupy), používame priame testy.

2. **Hranice medzi blokmi** – overuje sa komponenta, ako aj jej prepojenie s okolitými komponentami. Často sa totiž stáva, že každý z implementátorov pochopil špecifikáciu rozhrania inak. Nastavenie signálov na rozhraní sa testuje pomocou assertions (podmienky, ktoré musia byť vždy splnené, dané protokolom), napr. hodnoty signálov v špecifických časových okamihoch, vzájomné vzťahy signálov (handshaking, prechody medzi stavmi). Assertions nekontrolujú správnosť dát, ich modifikáciu, ani stratu. Požívajú sa priame testy aj automaticky generované transakcie.

3. **Testovanie celkového systému** – testujú sa dátové prepojenia a časovanie signálov, používajú sa automaticky generované transakcie.

2.6 Rozhrania a Assertions

Rozhrania sú konštrukcie pre vhodné zoskupenie súvisiacich signálov, typicky zbernicových. Dovoľujú definovať smer týchto signálov použitím modportov. Väčšinou sa vytvárajú tri modporty – vstupný, výstupný a kompletný pasívny modport (všetky signály sú vstupné) pre monitorovacie účely. Rozhrania umožňujú tiež špecifikovať časovanie použitím clocking blokov.

Verifikácia založená na assertions (ABS – Assertion-based Verification) mení tradičný návrhový proces, pretože táto metodológia pomáha formálne charakterizovať obsah návrhu a očakávané operácie. SystemVerilog má vlastný špecifický jazyk pre definíciu assertions, ktoré sú jedným z najužitočnejších verifikačných prostriedkov. Poskytujú efektívny spôsob

pri zisťovaní a lokalizácii chýb v návrhu umiestnením funkčných kontrol do kritických bodov návrhu a na interné ako aj externé rozhrania. Ide o výroky v temporálnej logike, ktoré sú väčšinou dané protokolom rozhrania a musia byť vždy pravdivé.

Keď assertion zlyhá počas simulácie, príčina je oveľa ľahšie identifikovateľná ako pri hľadaní v simulačnom priebehu.

2.7 Návrh testov – tvorba transakcií

Tradičné plánovanie verifikačného postupu zahŕňa identifikáciu cieľov testovania, najmä častých problémových oblastí.

Rozlišujeme dva postupy pri tvorbe testov designu (spôsoby tvorby transakcií):

1. **Priame testy** – ich tvorba je založená na znalostiach kľúčových funkcií hardwarového návrhu, overuje sa ich správnosť. Nie sú vhodné pre kompletné otestovanie súčinnosti medzi blokmi, či funkcionality celého systému (vytvorenie kompletnej sady priamych testov by bolo časovo náročné). Na verifikátorom zadané vstupy sa v simulácii produkujú konkrétne výstupy. Objavia sa bežné chyby, ktoré verifikátor dokáže na základe chovania systému predvídať.

2. **Testy automatickým generovaním transakcií** – odhaľujú nečakané chyby. K dispozícii je generátor transakcií, ktoré považujeme za náhodné, aj keď v skutočnosti je definovaný ich formát (napr. veľkosť transakcie) prostredníctvom constrains. Overujeme, či dáta takto vygenerovaných transakcií prechádzajú systémom v nezmenenej forme (nezávisle na počte vstupných, výstupných rozhraní), teda či nedochádza k ich strate alebo modifikácii.

Pri testovaní musíme brať ohľad na viacero faktorov. Ich vhodnou kombináciou vznikne sada testov, ktorej cieľom je plne pokryť funkcionalitu daného designu. Sú to:

a) *Kľúčové funkcie návrhu* – sú nevyhnutné k tomu, aby design plnil plánovanú funkciu (sú súčasťou špecifikačného dokumentu). Tvoria základ verifikácie, musia byť identifikované hneď na začiatku a musia byť kompletné, inak je verifikátor často nútený prerábať verifikačný plán a zasahovať do verifikačného prostredia, čo je zdĺhavé a implementačne náročné. Ich zoznam zostavujú návrhári systému, verifikátori a systémoví inžinieri (HW-SW codesign). Okrem kľúčových funkcií by mal byť známy zoznam chýb, z ktorých sa systém dokáže zotaviť a akým spôsobom, čo by sa malo tiež vo verifikácii overiť.

b) *Nastavenie generických parametrov* – najčastejšie ide o počet vstupných a výstupných rozhraní, či veľkosť bufferov. Návrh často podporuje viac konfigurácií nastavením generických parametrov. Pre každý parameter existuje množina hodnôt, ktoré mu môžu byť priradené. Úlohou verifikácie je otestovať všetky povolené kombinácie, je možné tiež vytvárať náhodne generované konfigurácie s obmedzujúcimi podmienkami pre povolené hodnoty parametrov(constrains).

c) *Formát dát* – väčšinou závisí od protokolu vstupného a výstupného rozhrania, ovplyvňuje veľkosť dát, ale aj napr. počet častí transakcie. Dáta reprezentujú vo verifikačnom systéme transakcie (automaticky generované alebo vytvárané priamo verifikátorom).

d) *Protokol* – definuje signály, pomocou ktorých je riadený tok a spracovanie dát. Určuje poradie nastavenia signálov a assertions.

e) *Delays, synchronizácia* – vkladanie čakacích stavov medzi transakcie a ich generovanie do systému. Ak je dodržaná synchronizácia, zariadenie funguje aj pri väčšom počte vstupných a výstupných rozhraní.

2.8 Pokrytie

Pokrytie (coverage) je meranie pokroku vo verifikačnom procese. Identifikuje testované a netestované časti návrhu. Rozlišujeme štyri základné kategórie pokrytia:

1. **Pokrytie kódu (code coverage)**
   - aká časť zdrojového kódu bola prevedená, ktorými z kódových konštrukcií bol tok programu ovplyvnený počas simulácie.
2. **Funkčné pokrytie (functional coverage)**
   - aká časť kľúčových funkcií designu bola preverená (pomocou cielených testovacích vzoriek transakcií),
   - do akej miery boli overené okrajové situácie, napr. generovanie transakcií maximálnej veľkosti, využívanie minimálnych oneskorení medzi generovanými transakciami, overenie chýb z ktorých sa systém dokáže zotaviť,
   - aké konfigurácie návrhu boli otestované (nastavenie generických parametrov),
   - aká časť z povolených kombinácií signálov na rozhraniach bola počas simulácie nastavená (command coverage).
3. **Pokrytie stavov (FSM coverage)**
   - špeciálna skupina pokrytia, obsahuje prvky pokrytia kódu ako aj funkčného pokrytia,
   - meranie navštívených stavov a prechodov v konečnom automate počas simulácie. Generované transakcie prechádzajúce systémom totiž spôsobujú, že signály postupne nadobúdajú rôzne stavy.

Nástroje pre meranie pokrytia zbierajú informácie počas simulácie a po jej ukončení produkujú správu o výsledkoch pokrytia. Úlohou verifikátora je priebežne sledovať celkové pokrytie a snažiť sa modifikovať existujúce testy (obmieňať veľkosť generovaných transakcií a parametrov), prípadne vytvárať nové. Výsledkom by malo byť až 100%-né pokrytie designu. Tieto zmeny môžu byť automatizované inteligentným programom. Tento postup sa označuje ako „coverage-driven verification“.

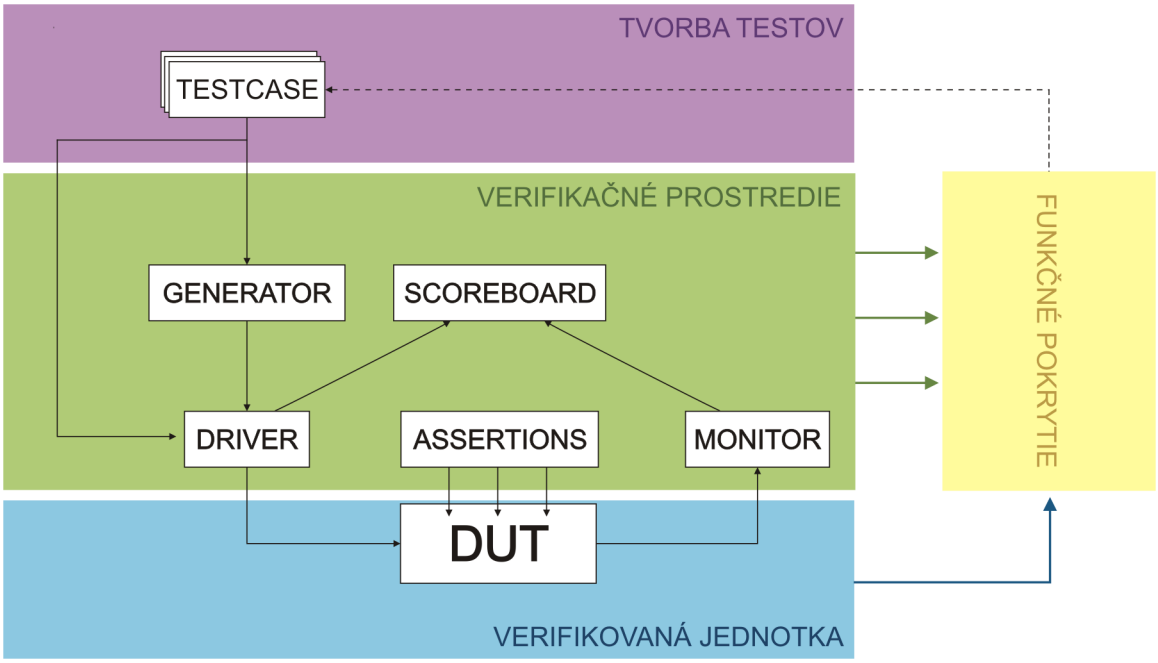
SystemVerilog umožňuje pridať špeciálny kód do verifikačného prostredia, ktorý sleduje command coverage. Vytvárame tzv. „cover groups“, kde definujeme monitorované signály a ich povolené kombinácie.

2.9 Tvorba verifikačného prostredia nad hardwarovou komponentou

Verifikačné prostredie pracuje na vysokej úrovni abstrakcie. Pre hardwarovú komponentu (DUT – Design Under Test) simuluje reálne prostredie, generuje a odchytáva transakcie

do/z DUT. Transakcie zodpovedajú protokolu rozhrania, väčšinou použitej zbernice napr. PCI, PCI Express, USB, Internal Bus, FrameLink atď.

Je výhodnejšie rozdeliť verifikačné prostredie na viac blokov, ktoré plnia špecifické funkcie. Návrh blokov vidno na obrázku 2.2. Tieto bloky sa vytvárajú na začiatku verifikácie, priebežne sa môžu vyvíjať pridávaním novej alebo vylepšenej funkcionality, ale počas testovania sú nemenné.



Obr. 2.2: Verifikačná schéma.

GENERATOR. Instancie generátora (ich počet je rovný počtu vstupných rozhraní DUT) produkujú náhodné transakcie. Ich formát je daný tzv. „blueprint“ transakciou, ktorá môže byť definovaná externe prostredníctvom formátovacích podmienok (constrains), vytvorených na základe protokolu rozhrania použitej zbernice. Kedykoľvek je potrebné zmeniť formát generovanej transakcie, stačí upraviť constrains blueprint transakcie. Generované transakcie sú následne ukladané do mailboxov.

DRIVER. Počet instancií drivera je opäť rovný počtu vstupných rozhraní DUT. Driver prijíma transakcie z príslušného mailboxu a zasiela ich do DUT cez rozhranie pripojenej zbernice. Za týmto účelom dochádza k prevodu z objektovej reprezentácie transakcie do signálovej reprezentácie danej zbernice. Po úspešnom odoslaní do DUT je možné zavolať špeciálnu metódu (callback), ktorá kópiu objektovej transakcie odošle do scoreboardu. Bude použitá neskôr pri meraní výsledného pokrytia. Callback metódu je možné použiť ešte pred odoslaním transakcie do DUT, čo umožní v prípade potreby modifikovať vygenerovanú transakciu.

MONITOR. Monitor je tiež pripojený k rozhraniu zbernice, ale tento krát na prijímacej strane komunikácie. Počet instancií je rovný počtu výstupných rozhraní DUT. Každá z instancií monitoruje cieľový port a transformuje signálovú reprezentáciu transakcie na objektovú. Po úspešnom prijatí je kópia objektovej transakcie odoslaná do scoreboardu (cez metódu callback).

SCOREBOARD. Zhromažďuje objektové transakcie, ktoré vstupujú do DUT (ich kópia je do scoreboardu odoslaná driverom). Implementuje routovací algoritmus. Po prijatí transakciu buď zahadzuje, alebo ukladá do transakčnej tabuľky. V špecifických prípadoch je možné meniť jej formát i transformovať dátový obsah. Transakcie, ktoré z DUT vystupujú, sú taktiež po prijatí monitorom odosielané do scoreboardu. Dochádza k ich porovnaniu s tými, ktoré existujú v transakčnej tabuľke a pri zhode sú z nej odstránené. Inak je reportovaná chyba. Overuje sa tak bezstratovosť a integrita dát pri prechode systémom a správnosť návrhu. Efektívne odhaľuje chyby v dátových výpočtoch, transformáciách a poradí. Je možné vďaka nemu identifikovať chýbajúce alebo sporné dáta. Používajú sa dva postupy pri porovnávaní položiek v transakčnej tabuľke:

1. **FIFO porovnanie** – kontrola poradia prichádzajúcich a odchádzajúcich transakcií. Transakcia, ktorá vystúpi z DUT je v scoreboarde porovnávaná s prvou transakciou, ktorá je uložená v transakčnej tabuľke. Ak nie sú zhodné, došlo k chybe.
2. **Porovnanie priechodom tabuľky** – transakcia, ktorá vystúpi z DUT je v scoreboarde porovnávaná postupne so všetkými transakciami v transakčnej tabuľke. Ak sa nenájde zodpovedajúca transakcia, došlo k chybe.

Kapitola 3

DMA radič

3.1 DMA prenos

Pri priamom riadení vstupno-výstupných operácií procesorom je rýchlosť prenosu dát medzi radičom periférneho zariadenia a operačnou pamäťou limitovaná rýchlosťou procesora. Každý elementárny prenos vyžaduje vykonanie niekoľkých inštrukcií, čo spôsobí viacero prístupov do pamäte, ktoré zahŕňajú okrem vlastného čítania a zápisu prenášaných dát aj určitú réžiu. Tento režim činnosti nazývame *PIO (Programmed Input/Output)*. Dáta sú prenášané trojfázovo – najskôr z registra periférneho zariadenia do univerzálneho registra procesoru a odtiaľ do operačnej pamäti (a naopak). Je to z toho dôvodu, že na zbernici nemôže byť vystavená adresa oboch komunikujúcich strán naraz.

Pri prenose bloku dát sa pritom využívajú pomerne jednoduché operácie, ktoré je možné realizovať aj pomocou špecializovaných hardwarových prostriedkov – DMA radiča. V nasledujúcom texte bude popísaná pôvodná koncepcia DMA prenosov na základe informácií z [5, 8, 1], ktorá sa v tejto podobe už v súčasnosti nepoužíva. Princíp je ale podobný i v dnešných systémoch s DMA.

DMA (Direct Memory Access – priamy prístup do pamäte) – režim práce, pri ktorom sa dáta prenášajú z dátového registra hardwarového subsystému (napr. radiča periférneho zariadenia) cez zbernicu priamo do operačnej pamäte (prípadne v opačnom smere) bez účasti procesora. Prenos je riadený radičom DMA a nesúvisí s realizáciou inštrukcie, čiže nie je vyvolaný vstupno-výstupnou inštrukciou.

Priamy prístup do pamäte je využívaný v situáciách, kedy je potrebné dáta prenášať do/z pamäte na vyšších rýchlostiach ako pri prenose riadenom procesorom. Ide najmä o diskové operácie, kde sú požiadavky na rýchlosť najväčšie (napr. v pevných diskoch, sieťových zariadeniach, CD/DVD mechanikách).

Radič DMA je automat, ktorý je schopný generovať riadiace signály zbernice podobne ako radič zbernice. Riadi prenos dát, samotné dáta ním neprechádzajú. Okrem riadiacich a stavových registrov obsahuje ďalšie dva registre, ktoré sú pre vlastný prenos podstatné: adresový register a register dĺžky prenosu. Tieto registre sú programovo prístupné a musia byť pri inicializácii DMA prenosu naplnené hodnotami, ktoré charakterizujú prenos.

- Adresový register – adresa pamäti, do/z ktorej bude prenos prebiehať.
- Register dĺžky prenosu – veľkosť bloku dát, ktorý bude prenášaný.

Zdroj prenosu je adresovaný ešte pred zahájením prenosu. Radič DMA je schopný spracovávať žiadosti o DMA prenos od viacerých perifémych zariadení, ich počet je však obmedzený

počtom kanálov radiča, kde každý kanál je vyhradený práve jednému periférnemu zariadeniu. Kanál tvoria dva vodiče. Prvým z nich žiada radič periférneho zariadenia o prenos a druhým je táto žiadosť radičom DMA potvrdená.

Procesor inicializuje prenos, ale už ho sám nevykonáva. V dobe, kedy DMA radič pracuje, môže spracovávať iné inštrukcie, nie je zbytočne zamestnávaný (na rozdiel od PIO režimu). Samotnej DMA operácie sa zúčastňuje iba radič periférneho zariadenia a jeho registre, radič DMA, systémová zbernica a operačná pamäť.

3.2 Režimy DMA

1. **kradnutie cyklov**

   DMA radič v prípade požiadavky na prenos obsadí zbernicu a prevedie jeden elementárny prenos dát. Potom zbernicu opäť uvoľní. Na zbernici sa tak striedajú prenosové cykly procesora a prenosové cykly radiča DMA.

2. **blokový režim**

   DMA radič obsadí zbernicu na začiatku blokového prenosu a drží ju obsadenú tak dlho, pokiaľ má čo prenášať. Týmto spôsobom dokáže skutočne využiť maximálnu možnú rýchlosť prenosu dát po zbernici.

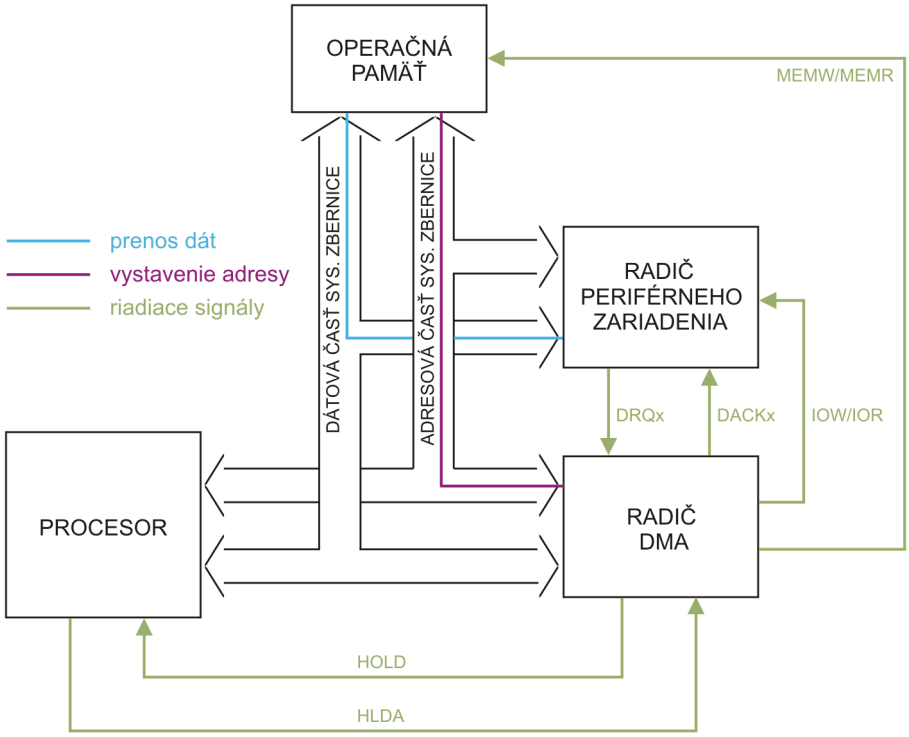
3.3 Činnosť DMA

Činnosť DMA je znázornená na obrázku 3.1.

Postupnosť krokov:

1. Programovo je prevedená inicializácia radiča PZ a DMA radiča.
2. Radič periférneho zariadenia má pripravené dáta, generuje signál DRQ x pre DMA prístup k zbernici.
3. Radič DMA zachytí žiadosť. Pred samotným prenosom musí požiadať procesor o pridelenie zbernice nastavením signálu HOLD. Procesor kladne odpovedá nastavením signálu HLDA, odpojí sa všetkými svojimi signálmi od zbernice. Svoje výstupy uvedie do stavu vysokej impedancie. Minimalizuje sa záťaž na zbernicu, ktorá bude od tejto chvíle riadená iba radičom DMA.
4. Radič DMA vyšle obsah adresového registra na adresovú časť systémovej zbernice a generuje riadiaci signál (čítanie/zápis z/do pamäti). Súčasne odpovedá radiču periférneho zariadenia potvrdením jeho žiadosti o prenos nastavením signálu DACK x, ktorý sa využíva aj pre adresáciu registra periférneho zariadenia, ktorý sa zúčastní prenosu dát.
5. Ako reakciu na signál DACK x začne radič PZ vysielať obsah dátového registra na dátovú časť systémovej zbernice.
6. Po ukončení prenosu sa generuje prerušenie, ktoré informuje procesor. Ten získa opäť právo riadiť a plne využívať zbernicu a pokračuje vo vykonávaní programu.

Niektoré DMA radiče sú vybavené dvoma adresovými registrami. Zdroj a cieľ dát tak môžu ležať v pamäťovom aj IO adresovom priestore.



Obr. 3.1: Princíp činnosti DMA radiča.

3.4 Vývoj DMA prenosu

Tento princíp začal byť široko využívaný u počítačov tretej generácie v 60.–70. rokoch minulého storočia.

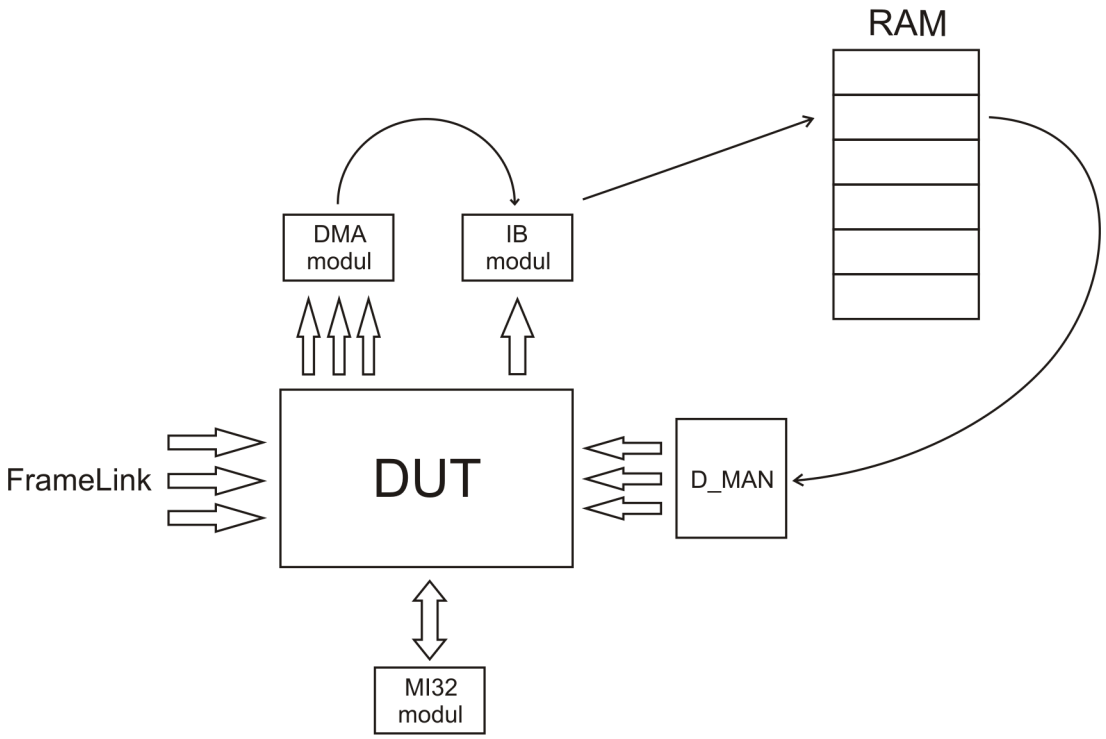
Najväčšie využitie zaznamenal v období počítačov založených na systémovej zbernici ISA. DMA radič je v tejto architektúre súčasťou čipovej sady. Má 16 kanálov, z toho 7 dostupných pre procesor počítača. Každý kanál má priradený 16-bitový adresový register a 16-bitový čítač. Pre zahájenie prenosu musí radič konkrétneho periférneho zariadenia nastaviť tieto registre a riadiaci signál pre zápis alebo čítanie. Následne DMA radič spracuje žiadosť o DMA prenos.

Zavedenie systémovej zbernice PCI viedlo k náhrade DMA bus-masteringom. Každé zariadenie, ktoré je schopné riadiť zbernicu, prevezme kontrolu a samostatne riadi prenos. Presnejšie PCI zariadenie požiada o pridelenie zbernice arbiter (severný most, tzv. supermaster) a ten rozhodne, ktoré zariadenie bude v danom okamihu bus-master. Kontrolu nad zbernicou môže mať totiž v danom okamihu iba jedno zariadenie.

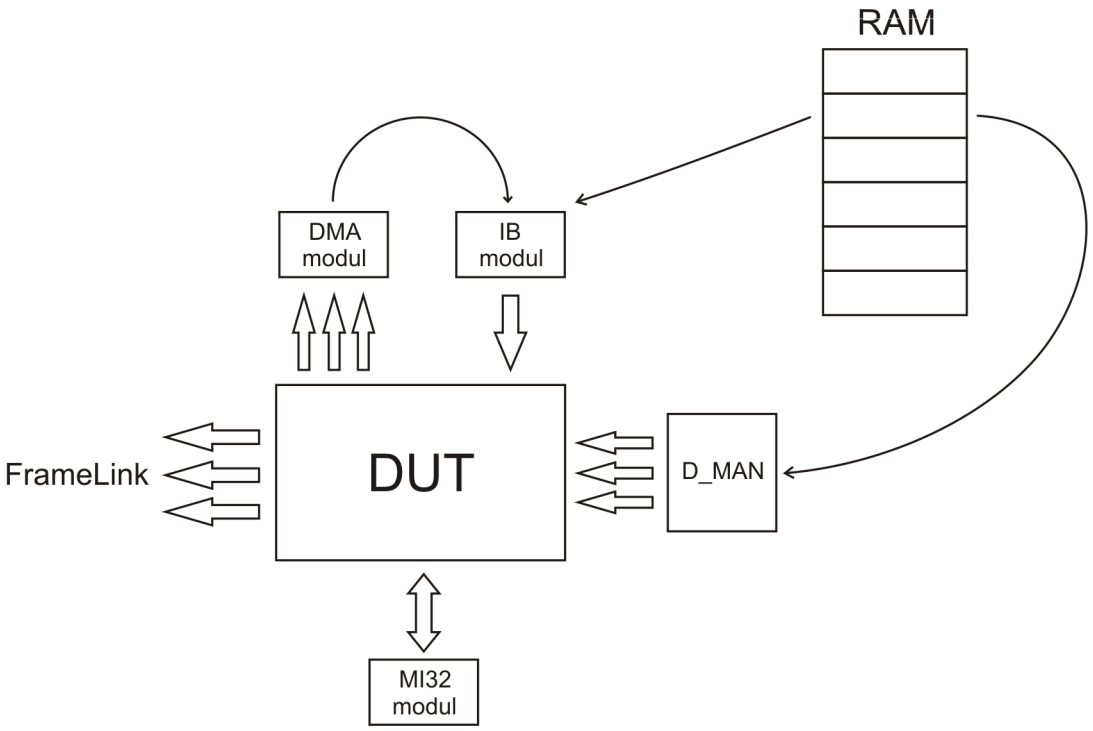
Kapitola 4

Návrh verifikačného prostredia DMA radičov

Testovaným designom sú dva typy DMA radičov. RX DMA radič (bloková schéma na obrázku 4.1) riadi prenos dát medzi hardwarovým a softwarovým bufferom a TX DMA radič (bloková schéma na obrázku 4.2) v opačnom smere. Radiče spolupracujú s viacerými komponentami v systéme. Sú to: DMA modul, modul riadiaci rozhranie internej zbernice (IB modul), operačná pamäť RAM, descriptor manager (D_MAN), modul riadiaci rozhranie MI32 (MI32 modul). Popis každého z nich bude obsahom nasledujúcich podkapitol. Zdrojom informácií o rozhraniach je Wiki projektu Liberouter [7].



Obr. 4.1: Bloková schéma RX DMA radiča.



Obr. 4.2: Bloková schéma TX DMA radiča.

4.1 Komponenty a rozhrania systému s radičmi

4.1.1 Rozhranie FrameLink

Transakcie vstupujúce do systému v RX smere a transakcie vystupujúce zo systému v TX smere sú definované protokolom rozhrania – FrameLinkom. Ide o protokol pre prenos dát vo forme paketov v projekte Liberouter. Je inšpirovaný protokolom LocalLink od Xilinxu. Signály rozhrania zobrazuje tabuľka 4.1.

Názov	**Smer RX**	**Smer TX**	**Dátová šírka**	**Popis**
DATA	in	out	DATA_WIDTH	Prenášané dáta.
REM	in	out	DREM_WIDTH	Platné bajty v poslednom slove.
SOF_N	in	out	1	Začiatok paketu.
EOF_N	in	out	1	Koniec paketu.
SOP_N	in	out	1	Začiatok časti paketu.
EOP_N	in	out	1	Koniec časti paketu.
SRC_RDY_N	in	out	1	Pripravenosť zdroja odosielať dáta.
DST_RDY_N	out	in	1	Pripravenosť cieľa prijať dáta.

Tabuľka 4.1: Rozhranie FrameLink.

Na základe protokolu FrameLink a časovej súčinnosti jednotlivých signálov je možné definovať **verifikačné požiadavky (requirements) pre FrameLinkové rozhranie**:

- Na dátovom porte DATA sú platné dáta, ak sú aktívne signály SRC_RDY_N a DST_RDY_N.

- SOF_N je aktívny zároveň s SOP (aktívne signály SRC_RDY_N a DST_RDY_N).
- EOF_N je aktívny zároveň s EOP_N (aktívne signály SRC_RDY_N a DST_RDY_N).
- S aktívnym EOP_N, SRC_RDY_N a DST_RDY_N je platný signál REM.
- Medzi EOF_N a SOF_N sa nevyskytujú dáta (SRC_RDY_N a DST_RDY_N nie sú aktívne súčasne).
- Po každom SOP_N musí po určitom čase nasledovať EOP_N.
- Po každom SOF_N musí po určitom čase nasledovať EOF_N.
- Signál SOF_N nesmie byť aktívny po predchádzajúcom SOF_N, pokiaľ medzi nimi nebol aktívny EOF_N.
- Signál SOP_N nesmie byť aktívny po predchádzajúcom SOP_N, pokiaľ medzi nimi nebol aktívny EOP_N.

Implementáciou requirementov rozhraní všetkých použitých modulov sú Assertions v SystemVerilogu. V tomto prípade som ich vytvorila pre vstupné aj výstupné FrameLinkové rozhranie.

4.1.2 Descriptor Manager

Komponenta, ktorá poskytuje radiču descriptory. Sú to adresy všetkých dátových blokov v RAM (4 kB pamäťové bloky – používajú sa ako softwarové dátové buffery), ktoré sú taktiež uložené v pamäti ako lineárny zoznam a zoskupené do 4 kB blokov. Radič ich používa pri prenosoch z/do fyzickej pamäte. Pred zahájením svojej činnosti musí byť tento modul správne nainicializovaný. Po inicializácii začne pripravovať descriptory a uchováva ich vo vnútornej pamäti až kým si ich radič nevyčíta cez fifo rozhranie (viď tabuľka 4.2).

Descriptor Manager pracuje so 64-bitovými descriptormi (horných 52 bitov použitých pre popis adresy, dolných 12 bitov pre ľubovoľné účely). Existujú dva typy descriptorov, ktoré sa odlišujú hodnotou v poslednom bite (LSB):

- typ 0 – klasický descriptor, ktorý odkazuje na dátový blok v RAM,
- typ 1 – odkazuje na miesto v RAM, kde sa nachádza ďalší blok descriptorov.

Názov	**Smer**	**Dátová šírka**	**Popis**
DESC_READ	out	1	Požiadavka na čítanie descriptora.
DESC_DO	in	CTRL_DMA_DATA_WIDTH	Descriptor.
DESC_EMPTY	in	1	Signalizácia pripravenosti nových dát.
DESC_ENABLE	out	1	Povoľovací signál pre radič.

Tabuľka 4.2: Descriptor Manager – FIFO rozhranie.

Verifikačné požiadavky pre rozhranie Descriptor managera:

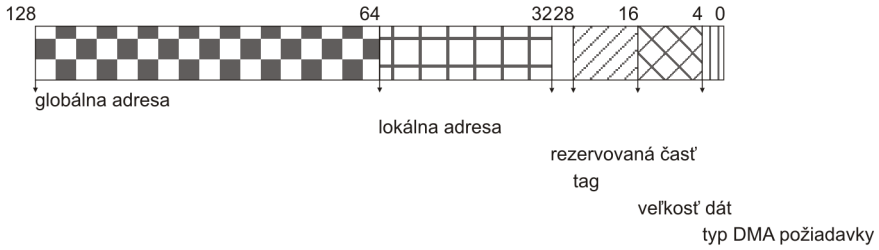
- Na dátovom porte DESC_DO sú platné dáta, ak sú aktívne signály DESC_READ a DESC_ENABLE a neaktívny signál DESC_EMPTY.
- Po každom DESC_READ musí po určitom čase nasledovať DESC_ENABLE.

4.1.3 DMA modul

Prijíma DMA požiadavky vytvárané radičom cez bus-masterové rozhranie. Signály rozhrania zobrazuje tabuľka 4.3. 128-bitová DMA požiadavka je postupne vyčítaná v niekoľkých taktoch. Jej formát vidno na obrázku 4.3. Požiadavky sú v DMA module rozparsované a uložené do mailboxu dmaMBX, z ktorého si ich postupne podľa potreby načítava IBUS modul.

Názov	**Smer**	**Dátová šírka**	**Popis**
DMA_ADDR	in	log2(128/DMA_DATA_WIDTH)	Adresa DMA požiadavky.
DMA_DOUT	out	DMA_DATA_WIDTH	DMA požiadavka.
DMA_REQ	out	1	Požiadavka na DMA prenos.
DMA_ACK	in	1	Potvrdenie DMA prenosu.
DMA_DONE	in	1	Ukončenie DMA prenosu.
DMA_TAG	in	1	Tag identifikujúci DMA požiadavku.

Tabuľka 4.3: DMA Bus-master rozhranie.



Obr. 4.3: 128 bitová DMA požiadavka rozčlenená na viacero logických častí.

Verifikačné požiadavky pre rozhranie DMA modulu:

- 1 takt po vystavení DMA_REQ nasleduje postupný prenos 128 bitovej DMA požiadavky (dátový port DMA_DOUT) so zodpovedajúcimi adresami na porte DMA_ADDR.
- Po každom DMA_REQ musí po určitom čase nasledovať DMA_ACK.
- Po každom DMA_ACK musí po určitom čase nasledovať DMA_DONE súčasne s DMA_TAG (môže potvrdzovať naraz viac DMA_ACK).
- Signál DMA_REQ nesmie byť aktívny po predchádzajúcom DMA_REQ, pokiaľ medzi nimi nebol aktívny signál DMA_ACK.
- Hodnota signálu DMA_TAG je platná ak je aktívny DMA_DONE.

4.1.4 IBUS modul

Modul rozhrania internej zbernice. Z mailboxu dmaMBX vyberá rozparsovanú DMA požiadavku, ktorá poskytuje informáciu o veľkosti dát prenášaných po internej zbernici, zdrojovú a cieľovú adresu prenosu. Na základe týchto údajov riadi prenos dát dvoma smermi:

- **Zápis dát do RAM** – RX smer, používa sa čítacie rozhranie internej zbernice viď tabuľka 4.4.
- **Čítanie dát z RAM** – TX smer, používa sa zápisové rozhranie internej zbernice viď tabuľka 4.5.

Názov	Smer	Dátová šírka	Popis
RD_ADDR	out	ADDR_WIDTH	Zdrojová adresa (HW buffer).
RD_BE	out	8	Byte enable.
RD_REQ	out	1	Požiadavka na čítanie z HW buffera.
RD_ARDY	in	1	Potvrdenie žiadosti o čítanie.
RD_DATA	in	DATA_WIDTH	Prenášané dáta.
RD_SRC_RDY	in	1	Zdroj poskytuje validné dáta.
RD_DST_RDY	out	1	Cieľ je schopný prijať dáta.

Tabuľka 4.4: Čítacie rozhranie internej zbernice.

Názov	Smer	Dátová šírka	Popis
WR_ADDR	out	ADDR_WIDTH	Cieľová adresa (HW buffer).
WR_BE	out	8	Byte enable.
WR_REQ	out	1	Požiadavka na zápis do HW buffera.
WR_RDY	in	1	Potvrdenie žiadosti o zápis.
WR_DATA	out	DATA_WIDTH	Prenášané dáta.

Tabuľka 4.5: Zápisové rozhranie internej zbernice.

Verifikačné požiadavky pre čítacie a zápisové rozhranie internej zbernice:

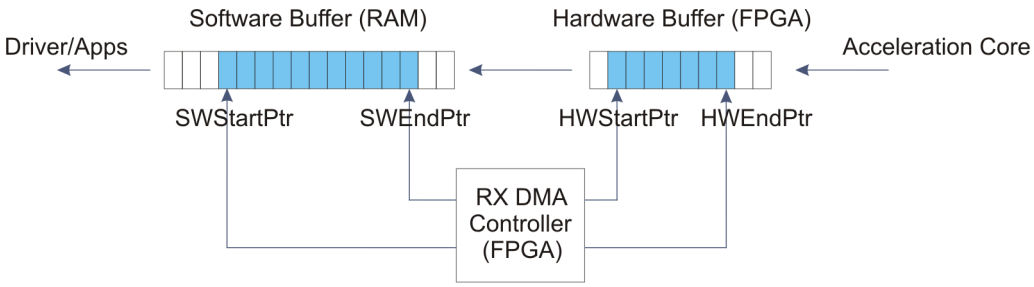
- Prenos cez čítacie rozhranie sa začína po súčasnej aktivite RD_REQ a RD_ARDY.
- Po RD_REQ musí po určitom čase nasledovať RD_SRC_RDY súčasne s RD_DST_RDY. V tomto okamihu sú na porte RD_DATA platné dáta a na porte RD_ADDR platná adresa.
- Signál RD_REQ nesmie byť aktívny po predchádzajúcom RD_REQ, pokiaľ medzi nimi nebol aktívny signál RD_SRC_RDY súčasne s RD_DST_RDY.
- Prenos cez zápisové rozhranie sa začína po súčasnej aktivite signálov WR_REQ a WR_RDY, na porte WR_DATA sú platné dáta a na porte WR_ADDR adresa.

4.1.5 MI32 modul

Rýchly prenos dát je založený na jednoduchom princípe prenosu medzi dvoma kruhovými buffermi – hardwarovým (na akceleračnej karte, DMA buffer) a softwarovým (pamäť RAM). Aby bolo možné identifikovať, koľko je v každom bufferi uložených dát, je pre každý buffer použitá vždy dvojica ukazateľov ukazujúcich na začiatok a koniec dát. Úlohou DMA radiča je kontrolovať hodnoty všetkých štyroch ukazateľov a inicializovať DMA prenos pokiaľ je to možné. Princíp nastavovania ukazateľov popisuje nasledujúci text a obrázky.

- **Prenos dát z DMA buffera do RAM (obrázok 4.4)**

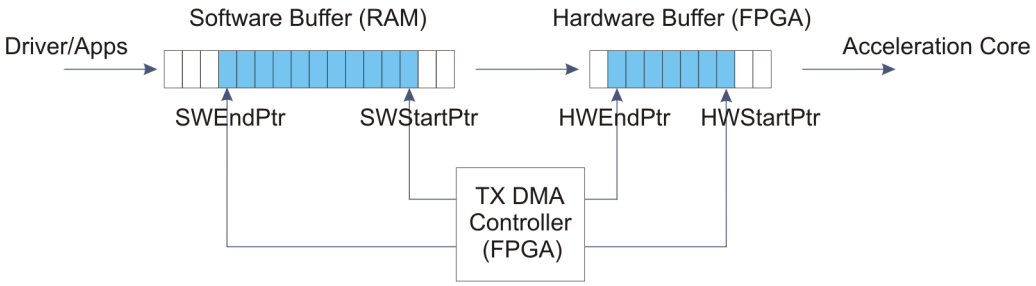
  1. Do vstupného DMA buffera sú uložené nové dáta pre prenos do RAM.
  2. Informácia o nových dátach je predaná DMA radiču. Ten posunie hodnotu HWEndPointer, skontroluje, či je pre nové dáta dostatok miesta v pamäti RAM (SWEndPointer - SWStartPointer $\geq$ NewDataSize). Pokiaľ áno, generuje DMA požiadavku pre prenos dát.
  3. Akonáhle je prenos dokončený, DMA radič posunie hodnotu ukazateľa SWEndPointer o prenesené dáta, predá DMA bufferu informáciu o veľkosti prenesených dát a buffer dáta uvoľní. Ďalej posunie hodnotu ukazateľa HWEndPointer na príslušnú pozíciu. Pokiaľ je to požadované, vygeneruje prerušenie informujúce o nových dátach.
  4. Softwarová aplikácia periodicky žiada Driver o nové dáta. Driver overí prítomnosť nových dát kontrolou oboch ukazateľov a pokiaľ má dáta k dispozícii, predá ich aplikácii. V opačnom prípade predá aplikácii chybový návratový kód alebo aplikáciu uspí a vyčká na príchod nových dát.
  5. Body 1-4 sa neustále opakujú.



Obr. 4.4: Prenos dát z DMA buffera do RAM.

- **Prenos dát z RAM do DMA buffera (obrázok 4.5)**

  1. Softwarová aplikácia vloží do buffera v RAM nové dáta a predá túto informáciu driveru.
  2. Driver predá DMA radiču informáciu o nových dátach posunutím hodnoty SWEndPointer na príslušnú pozíciu.
  3. DMA radič prevedie kontrolu, či je vo vysielacom DMA bufferi dostatok miesta pre prenos a pokiaľ áno, generuje DMA požiadavku.
  4. Po ukončení prenosu DMA radič posunie hodnotu ukazateľa SWStartPointer o prenesené dáta (dáta v RAM sa uvoľnia) a pokiaľ je to požadované, vygeneruje sa prerušenie informujúce o uvoľnenom mieste v pamäti RAM. Predá DMA bufferu informáciu o veľkosti prenesených dát a upraví hodnotu HWEndPointer.
  5. DMA buffer dáta odošle, predá DMA radiču informáciu o uvoľnenom mieste a DMA radič posunie hodnotu HWEndPointer na príslušnú pozíciu.
  6. Body 1–5 sa neustále opakujú.



Obr. 4.5: Prenos dát z RAM do DMA buffera.

Operačný systém zabezpečuje cez MI32 modul konfiguráciu radiča v dobe inicializácie, nastavenie a čítanie hodnôt SWStartPointer a SWEndPointer (cez rozhranie zobrazené v tabuľke 4.6). Riadi tak načítavanie a zápis dát do softwarového buffera – RAM.

Názov	**Smer**	**Dátová šírka**	**Popis**
SW_DWR	in	32	Zapisované dáta.
SW_ADDR	in	32	Zápisová adresa.
SW_RD	in	1	Požiadavka na čítanie.
SW_WR	in	1	Požiadavka na zápis.
SW_BE	in	4	Byte enable.
SW_DRD	out	32	Čítané dáta.
SW_ARDY	out	1	Pripravenosť na zmenu adresy.
SW_DRDY	out	1	Čítané dáta pripravené.

Tabuľka 4.6: Zápisové a čítacie rozhranie mi32.

Verifikačné požiadavky pre rozhranie MI32:

- S aktívnymi signálmi SW_WR a SW_ARDY sú na porte SW_DWR platné dáta a na porte SW_ADDR platná adresa.
- S aktívnymi signálmi SW_RD a SW_ARDY je na porte SW_ADDR platná adresa.
- S aktívnym signálom SW_DRDY sú na porte SW_DRD platné dáta.
- SW_RD a SW_WR nie sú aktívne súčasne.

4.1.6 RAM

Softwarový buffer (RAM) je zdrojom a cieľom prenosu dát. Vo verifikácii nepracujeme so skutočnou pamäťou, iba jej behaviorálnou simuláciou. Môžeme tak overiť správnosť zápisu na vybrané adresy, či konzistenciu dát. Tvorí ju pole o veľkosti:

N = počet vstupných rozhraní . počet stránok pamäte pre 1 rozhranie . veľkosť stránky v bajtoch

Transakcie sa zapisujú do položiek poľa a sú z nich tiež čítané.

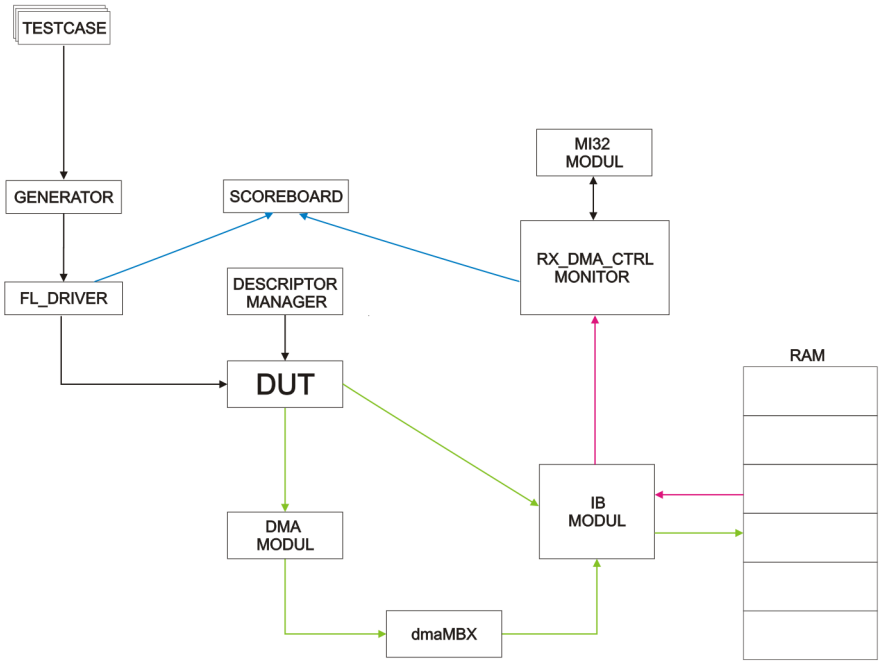
4.2 RX DMA radič

RX DMA radič riadi prenos dát z hardwarového buffera do softwarového.

1. Na začiatku dôjde k inicializácii DMA radiča prostredníctvom rozhrania MI32, ktorá odštartuje jeho činnosť.
2. Dáta, ktoré sú privedené na vstupné FrameLinkové rozhranie, sú po blokoch postupne ukladané do vstupného kruhového HW buffera. Po príchode nového bloku predá buffer informáciu o príchode nových dát radiču.
3. DMA radič vyhodnotí situáciu na základe štvorice ukazateľov do oboch bufferov. Pokiaľ je možné prenos uskutočniť, generuje DMA požiadavku, ktorá obsahuje adresu odosielateľa (lokálna adresa v HW bufferi), adresu príjemcu (globálna adresa do SW buffera) a veľkosť prenášaných dát.
4. Cez internú zbernicu sú dáta zapísané na cieľovú adresu do SW buffera. Radič aktualizuje hodnoty SW a HW ukazateľov, prípadne generuje prerušenie.
5. Radič potrebuje pri svojej činnosti poznať adresu blokov v RAM, kam budú dáta ukladané. Periodicky žiada Descriptor Manager o príslušné descriptory.

4.2.1 Verifikačné prostredie RX DMA radiča

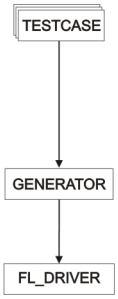
Pre verifikáciu RX DMA radiča som navrhla architektúru verifikačného prostredia (testbench) na obrázku 4.6. Šípky určujú smer toku dát.



Obr. 4.6: Verifikačné prostredie RX DMA radiča.

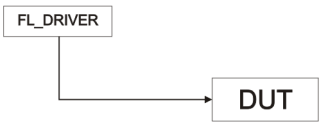
Nasledujúci text s obrázkami objasňuje funkciu komponent verifikačného prostredia.

1. časť: Testy pre overenie funkcionality RX DMA radiča. Tvorba priamych testov nie je v tomto prípade nevyhnutná, pretože jednotlivé komponenty prostredia už boli samostatne verifikované. Transakcie, ktoré do systému vstupujú, je výhodnejšie vytvárať automatickým generátorom transakcií. Formát každej z nich je vyhradený protokolom rozhrania – FrameLinkom. Produkuje sa toľko transakcií, aby celkové pokrytie systému viedlo k 100%. Viď obrázok 4.7.



Obr. 4.7: 1. časť verifikačného prostredia z obrázka 4.6.

2. časť: FrameLinkový driver prijíma vygenerované transakcie. Transformuje ich z objektovej reprezentácie na signálovú a odosiela smerom do verifikovanej komponenty (DUT) cez FrameLinkové rozhranie. Viď obrázok 4.8.

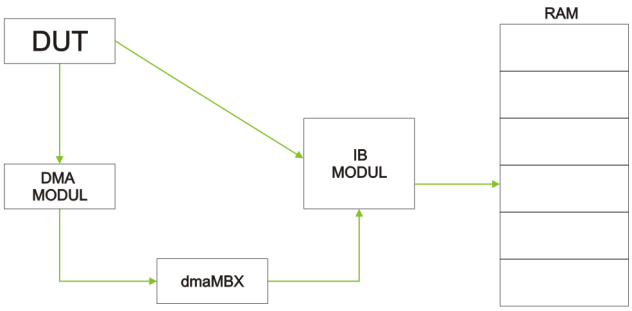


Obr. 4.8: 2. časť verifikačného prostredia z obrázka 4.6.

3. časť: Descriptor manager poskytuje v prípade potreby radiču descriptory pre všetky kanály (flows). Akonáhle si radič descriptor vyčíta, jednotka pripraví nový descriptor do fronty. Viď obrázok 4.9.

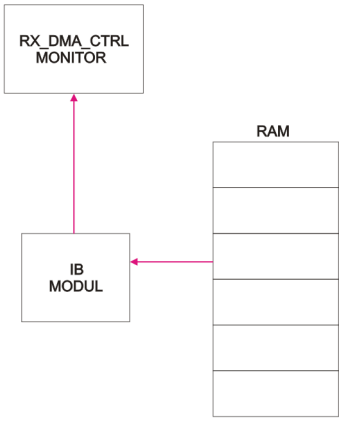
Obr. 4.9: 3. časť verifikačného prostredia z obrázka 4.6.

4. časť: DUT na základe prijatých dát generuje DMA požiadavky, ktoré sú cez DMA rozhranie prijaté DMA modulom. DMA Modul požiadavku rozdelí na viacej častí: lokálnu adresu, globálnu adresu, tag, veľkosť prenášaných dát a typ. V tomto tvare je informácia uložená do mailboxu dmaMBX. Modul riadiaci rozhranie internej zbernice (IB modul) vyberá položky z dmaMBX mailboxu a na základe získaných informácií riadi prenos dát z DUT cez internú zbernicu do pamäte (RAM). Viď obrázok 4.10.



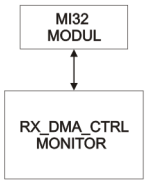
Obr. 4.10: 4. časť verifikačného prostredia z obrázka 4.6.

5. časť: RX_DMA_CTRL monitor overuje korektnosť dát zapísaných do pamäte. Priamo nesúvisí s činnosťou radiča, overuje či nedošlo k modifikácii alebo strate dát pri prenosoch. Za týmto účelom zisťuje zmenu SWEndPointera (prostredníctvom funkcie modulu MI32), ktorá sa mení každým zápisom do RAM. Na základe tejto informácie vyčíta dáta z presnej pozície v RAM. Prijaté dáta transformuje do protokolu FrameLink. Viď obrázok 4.11.



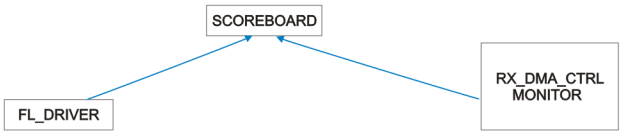
Obr. 4.11: 5. časť verifikačného prostredia z obrázka 4.6.

6. časť: MI32 modul zabezpečuje konfiguráciu v dobe inicializácie. Obsahuje tiež funkcie pre zápis a čítanie SW ukazateľov, ktoré používa RX_DMA_CTRL monitor pri prenose dát z pamäte RAM. Viď obrázok 4.12.



Obr. 4.12: 6. časť verifikačného prostredia z obrázka 4.6.

7. časť: FL_driver zasiela callbackom FrameLinkové transakcie do scoreboardu, v ktorom zapĺňajú transakčnú tabuľku. Transakcie prijaté RX_DMA_CTRL monitorom sú taktiež odosielané callbackom do scoreboardu a následne porovnávané princípom FIFO s tými, ktoré sú uložené v transakčnej tabuľke. Pri zhode sú z transakčnej tabuľky odstránené. Akonáhle v ňom niektoré z nich zostanú, indikuje to chybu pri prenose dát, vo funkcii radiča alebo v moduloch. Viď obrázok 4.13.



Obr. 4.13: 7. časť verifikačného prostredia z obrázka 4.6.

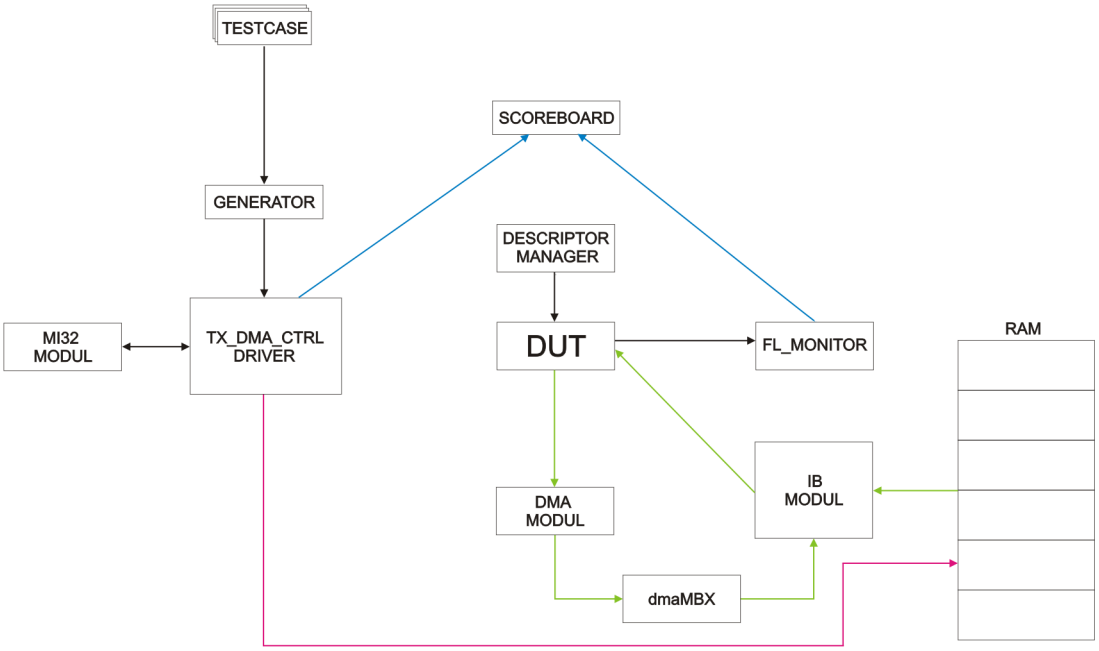
4.3 TX DMA radič

TX DMA radič riadi prenos dát zo softwarového buffera do hardwarového.

1. Na začiatku dôjde k inicializácii DMA radiča prostredníctvom rozhrania MI32, ktorá odštartuje jeho činnosť.
2. Na základe zmeny SW ukazateľov (súvisí so zápisom dát do kruhového SW buffera) je DMA radič informovaný o príchode nových dát.
3. Pokiaľ je možné prenos uskutočniť (kontrola voľného miesta v HW bufferi), radič vygeneruje DMA požiadavku, ktorá obsahuje adresu odosielateľa (globálna adresa v SW bufferi), adresu príjemcu (lokálna adresa do HW buffera) a tiež veľkosť prenášaných dát.
4. Cez internú zbernicu sú dáta vyčítané zo SW buffera a následne prenášané do HW buffera. Po uskutočnení prenosu radič aktualizuje hodnoty SW a HW ukazateľov, prípadne generuje prerušenie.
5. Radič vystaví dáta na výstupné FrameLinkové rozhranie.
6. Radič potrebuje pri svojej činnosti poznať adresu blokov v RAM, odkiaľ budú dáta čítané. Periodicky žiada Descriptor Manager o príslušné descriptory.

4.3.1 Verifikačné prostredie TX DMA radiča

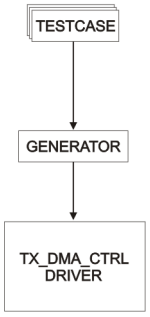
Pre verifikáciu TX DMA radiča som navrhla architektúru verifikačného prostredia (testbench) na obrázku 4.14. Šípky určujú smer toku dát.



Obr. 4.14: Verifikačné prostredie TX DMA radiča.

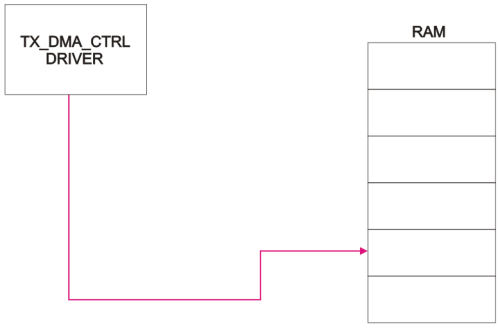
Nasledujúci text s obrázkami objasňuje funkciu komponent verifikačného prostredia.

1. časť: Testy pre overenie funkcionality TX DMA radiča. Podobne ako v RX radiči, transakcie, ktoré do systému vstupujú, je výhodnejšie vytvárať automatickým generátorom transakcií. Formát každej z nich je vyhradený protokolom rozhrania – FrameLinkom. Produkuje sa toľko transakcií, aby celkové pokrytie systému viedlo k 100%. Viď obrázok 4.15.



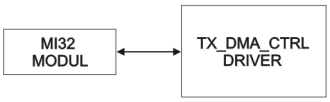
Obr. 4.15: 1. časť verifikačného prostredia z obrázka 4.14.

2. časť: TX_DMA_CTRL driver prijíma vygenerované transakcie a transformuje ich z formátu FrameLink do formátu internej zbernice. Ukladá ich do RAM na adresu, na ktorú ukazuje SWEndPointer. Následne nastaví novú hodnotu SWEndPointera. Viď obrázok 4.16.



Obr. 4.16: 2. časť verifikačného prostredia z obrázka 4.14.

3. časť: MI32 modul zabezpečuje konfiguráciu v dobe inicializácie. Obsahuje tiež funkcie pre zápis a čítanie SW ukazateľov, ktoré používa TX_DMA_CTRL driver pri prenose dát do pamäte RAM. Viď obrázok 4.17.

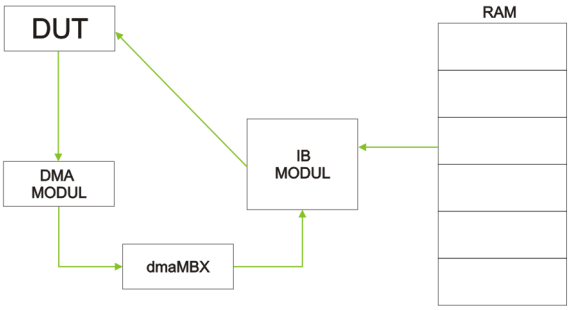


Obr. 4.17: 3. časť verifikačného prostredia z obrázka 4.14.

4. časť: Descriptor manager poskytuje v prípade potreby radiču descriptory pre všetky kanály (flows). Akonáhle si radič descriptor vyčíta, jednotka pripraví nový descriptor do fronty. Viď obrázok 4.18.

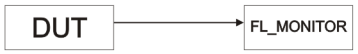
Obr. 4.18: 4. časť verifikačného prostredia z obrázka 4.14.

5. časť: DUT na základe zmeny SWEndPointera (po zápise dát do RAM) generuje DMA požiadavky, ktoré sú cez DMA rozhranie prijaté DMA modulom. DMA Modul požiadavku rozdelí na viacej častí: lokálnu adresu, globálnu adresu, tag, veľkosť prenášaných dát a typ. V tomto tvare je informácia uložená do mailboxu dmaMBX. Modul riadiaci rozhranie internej zbernice (IB modul) vyberá položky z dmaMBX mailboxu a na základe získaných informácií riadi prenos dát z RAM do DUT cez internú zbernicu. Viď obrázok 4.19.



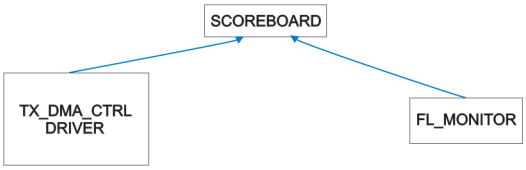
Obr. 4.19: 5. časť verifikačného prostredia z obrázka 4.14.

6. časť: FrameLinkový monitor sleduje výstupné FrameLinkové rozhranie DUT a transformuje prijímanú transakciu zo signálovej reprezentácie do objektovej. Viď obrázok 4.20.



Obr. 4.20: 6. časť verifikačného prostredia z obrázka 4.14.

7. časť: TX_DMA_CTRL zasiela callbackom FrameLinkové transakcie do scoreboardu, v ktorom zapĺňajú transakčnú tabuľku. Transakcie prijaté FrameLinkovým monitorom sú taktiež odosielané callbackom do scoreboardu a následne porovnávané princípom FIFO s tými, ktoré sú uložené v transakčnej tabuľke. Pri zhode sú z transakčnej tabuľky odstránené. Akonáhle v ňom niektoré z nich zostanú, indikuje to chybu pri prenose dát, vo funkcii radiča alebo v moduloch. Viď obrázok 4.21.



Obr. 4.21: 7. časť verifikačného prostredia z obrázka 4.14.

Kapitola 5

Implementácia a analýza výsledkov

5.1 RX DMA radič

Prechod transakcie verifikačným systémom RX DMA radiča je podrobne popísaný v Prílohe A. Pri implementácii prostredia som pracovala s referenčnou príručkou SystemVerilogu [4].

5.1.1 Implementácia – popis tried

1. **FrameLinkTransaction.** Táto trieda definuje formát FrameLinkových transakcií (paketov). Umožňuje napevno nastaviť počet častí paketu (ohraničené SOP, EOP) a rozsah, v ktorom sa bude dynamicky generovať veľkosť dát prenášaných v pakete. Dáta sa generujú náhodne. Obsahuje tiež metódy pre výpis (`display()`), kopírovanie (`copy()`) a porovnávanie (`compare()`) FrameLinkových transakcií.

2. **FrameLinkDriver.** Každá instancia tejto triedy je pripojená na jedno FrameLinkové rozhranie a jeden mailbox. Mailbox posúva FrameLinkové transakcie driveru. Driver ich transformuje do signálovej reprezentácie a odosiela cez FrameLinkové rozhranie do DUT. Po odoslaní sa kópia objektovej transakcie zasiela prostredníctvom callback funkcie do scoreboardu, kde bude neskôr použitá pri vyhodnotení celkového pokrytia.

3. **RxDescManager.** Pripravuje descriptory do fronty pre jednotlivé kanály, pričom aktuálny descriptor zasiela podľa potreby na príslušné rozhranie (`sendDescriptor()`). Akonáhle si DUT descriptor vyčíta (nastavením signálu DESC_READ), pripraví sa do fronty nový descriptor (`addDescriptor()`).

4. **RxDmaModul.** Prijíma radičom generované DMA požiadavky z bus-masterového rozhrania (`receiveDMA()`) a vytvára z nich štruktúrovanú informáciu (`parseDma()`), ktorá obsahuje nasledujúce časti: globálna adresa do RAM, lokálna adresa do HW buffera, veľkosť prenášaných dát a tag. V takomto tvare je informácia uložená do mailboxu dmaMbx (`add()`).

5. **RxIbusModul.** Vyberá rozčlenenú DMA požiadavku z mailboxu dmaMbx aby získal potrebné údaje na prenos medzi HW bufferom a RAM. Na základe informácie o veľkosti prenášaných dát vyčíta dáta z DUT cez čítacie rozhranie internej zbernice (`setIbus()`). Získané dáta zapíše na globálnu adresu do RAM.

6. **RxSoftwareModul.** Obsahuje metódy pre inicializáciu DMA radičov (`initCtrl()`), nastavenie SWStartPointera (`setStrPtr()`) a čítanie SWEndPointera (`getEndPtr()`)

prostredníctvom softwarového rozhrania MI32.

7. **RxDmaCtrlMonitor.** Detekuje zmenu SWEndPointera (`detectEndPointerChange()`) a na základe získaných údajov vyčítava dáta z RAM (`receiveTransaction()`). Po úspešnom prijatí ich transformuje do objektového formátu FrameLinkovej transakcie, ktorú zasiela prostredníctvom callback funkcie do scoreboardu.

8. **Scoreboard.** Táto trieda zhromažďuje framelinkové transakcie zasielané callback funkciou z drivera (trieda ScoreboardDriverCbs) a z monitora (trieda ScoreboardMonitorCbs). ScoreboardDriverCbs ukladá transakcie do ScoreboardTable. ScoreboardMonitorCbs hľadá zhodu medzi prijatou transakciou a transakciou v ScoreboardTable a keď ju nájde, vymaže ju. Kvôli konkurentnému prístupu z viacerých instancií drivera a monitora do ScoreboardTable je implementovaný synchronizačný mechanizmus – semafor.

9. **Coverage.** Trieda Coverage obsahuje niekoľko metód, ktoré agregujú informácie o pokrytí z rôznych coverage groups pre jednotlivé rozhrania RX DMA radiča. Získame tak prehľad o nastavení jednotlivých signálov a ich kombinácií počas simulácie.

5.1.2 Návrh testov, nastavenie generických parametrov

Pre overenie funkcionality RX DMA radiča som navrhla testovanie prostredníctvom automatického generovania obsahovo náhodných transakcií, ktorých formát je daný protokolom vstupného rozhrania – FrameLinkom.

Pred testovaním je potrebné nastaviť generické parametre radiča (tabuľka 5.1):

Názov	Popis	Hodnoty
BUFFER_DATA_WIDTH	Dátová šírka RX buffera.	64
BUFFER_BLOCK_SIZE	Maximálny počet položiek v bloku RX buffera.	512
BUFFER_FLOWS	Počet kanálov.	1/2/4
BUFFER_TOTAL_FLOW_SIZE	Veľkosť RX buffera v bajtoch pre jeden kanál.	8192
CTRL_BUFFER_ADDR	Bázová adresa RX buffera.	0
CTRL_DMA_DATA_WIDTH	Dátová šírka DMA požiadavky.	8/16
PACKET_COUNT	Počet častí paketu.	2
PACKET_SIZE_MAX	Horná hranica pri generovaní veľkosti paketu.	1. paket 32, 2. paket 1530
PACKET_SIZE_MIN	Dolná hranica pri generovaní veľkosti paketu.	1. paket 0, 2. paket 1

Tabuľka 5.1: Generické parametre.

Na základe rôznych kombinácií generických parametrov som zostavila testy znázornené v tabuľke 5.2. Nastavenie kombinácií v ModelSime zobrazuje v prílohe obrázok B.1.

Generiky	Test 1		Test 2		Test 3		Test 4		Test 5		Test 6	
BUFFER_DATA_WIDTH	64		64		64		64		64		64	
BUFFER_BLOCK_SIZE	512		512		512		512		512		512	
BUFFER_FLOWS	1		1		2		2		4		4	
BUFFER_TOTAL_FLOW_SIZE	8192		8192		8192		8192		8192		8192	
CTRL_BUFFER_ADDR	0		0		0		0		0		0	
CTRL_DMA_DATA_WIDTH	8		16		8		16		8		16	
PACKET_COUNT	2		2		2		2		2		2	
PACKET_SIZE_MAX	32	1530	32	1530	32	1530	32	1530	32	1530	32	1530
PACKET_SIZE_MIN	0	1	0	1	0	1	0	1	0	1	0	1

Tabuľka 5.2: Návrh testov pre RX DMA radič.

5.1.3 Analýza chybových stavov

Stavy, kedy nastavenie signálov na jednom alebo viacerých rozhraniach nezodpovedá špecifikácii, alebo komponenta produkuje chybné výstupy, je možné označiť ako chybové stavy. Simulačné prostredie v ModelSime mi výrazne pomohlo pri ich analýze a oprave. Nasleduje podrobná analýza jednej z objavených chýb v RX DMA radiči a postupnosť krokov pri jej hľadaní.

Krok 1. Chyba sa typicky prejaví nezhodou v transakčnej tabuľke, čo je možné vidieť na obrázku 5.1. Vidíme, že na vstup monitora prichádza neznáma transakcia. Presný dátový obsah transakcie, ktorá sa mala na výstupe objaviť, je zrejmý z výpisu pod transakčnou tabuľkou. Ich porovnaním zistíme, že veľkosť prijatých dát je správna (veľkosť prvej časti paketu je 32, druhej časti paketu 2), ale časť transakcie obsahuje namiesto platných dát nuly.

```
# Unknown transaction received from monitor Monitor0
# Time: 11705,000 ns
# Ifc: 0
# Packet no:          0, Packet size:         32, Data: feaf2b84d4f5089312e6a8d9e289947552b59dbc4706755d0000000000000000
# Packet no:          1, Packet size:          2, Data: 0000
# ------------------------------------------------------------
# --  TRANSACTION TABLE
# ------------------------------------------------------------
# Size:          39
# Items added:        146
# Items removed:        107
# Ifc: 0
# Packet no:          0, Packet size:         32, Data: feaf2b84d4f5089312e6a8d9e289947552b59dbc4706755d818017d688ce17c2
# Packet no:          1, Packet size:          2, Data: 36f4
```

Obr. 5.1: Na výpise z ModelSimu je možné vidieť nezhodu v transakčnej tabuľke. Na výstup monitoru prichádza neznáma transakcia.

Krok 2. Akonáhle sa objaví tento typ problému, je nutné odsledovať priechod transakcie systémom v nasledujúcich krokoch:

a) Priechod transakcie FrameLinkovým rozhraním, výpis z FrameLinkového drivera.

b) Priechod transakcie rozhraním internej zbernice, kontrola DMA požiadavky.

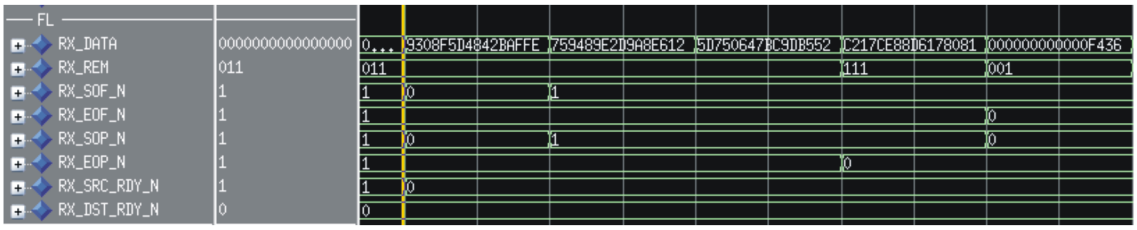
c) Uloženie transakcie do RAM, kontrola adries, na ktoré sa zapisuje.

d) Čítanie transakcie z RAM, výpis z monitora.

Krok 3. Na obrázku 5.2 je vidieť výpis z FrameLinkového drivera a na obrázku 5.3 priechod transakcie (paketu) cez FrameLinkové rozhranie. Dátový obsah transakcie je korektný, signály sú nastavené správne. Paket sa skladá z dvoch častí, ktoré sú ohraničené signálmi RX_SOP_N a RX_EOP_N. Celkovo je paket ohraničený signálmi RX_SOF_N a RX_EOF_N. Hodnota signálu RX_REM určuje, koľko bajtov dát je platných v poslednom slove v jednotlivých častiach paketu . V poslednom slove prvej časti paketu sú platné všetky bajty (hodnota RX_REM je 111), v druhej časti paketu iba 2 bajty (hodnota RX_REM je 001). Na FrameLinku nedochádza k strate dát.

```
# ------------------------------------------------------------
# -- DRIVER
# ------------------------------------------------------------
# Ifc: 0
# Packet no:          0, Packet size:         32, Data: feaf2b84d4f5089312e6a8d9e289947552b59dbc4706755d818017d688ce17c2
# Packet no:          1, Packet size:          2, Data: 36f4
```

Obr. 5.2: Výpis z FrameLinkového drivera, vidno dátový obsah transakcie.



Obr. 5.3: Priechod transakcie FrameLinkovým rozhraním. Nedochádza k zmene dátového obsahu ani chybe v nastavení signálov.

Krok 4. Obrázok 5.4 zobrazuje priechod transakcie internou zbernicou. Dátový obsah je opäť správny, vrátane nastavenia jednotlivých signálov.

Obr. 5.4: Priechod transakcie rozhraním internej zbernice. Nedochádza k zmene dátového obsahu ani chybe v nastavení signálov.

Krok 5. Výpis z IB modulu (obrázok 5.5) zobrazuje, aké dáta sú ukladané do RAM a na akú adresu. Je zrejmé, že transakcia sa zapisuje do oblasti, kde dochádza k prechodu medzi stránkami pamäte (veľkosť jednej stránky je 4096 bajtov, čiže 512 64-bitových dátových položiek), konkrétne na adresy 509, 510, 511, 512, 513. Práve v tomto bode mohlo dôjsť ku konfliktu. Vzniká tak podozrenie na chybnú funkciu SW ukazateľov. Vysvetlilo by to problém so stratou časti dát transakcie.

```
# IB MODUL: Adresa pri zapise do RAM:                509
# IB MODUL: Zapisovane data do RAM: 9308f5d4842baffe
# IB MODUL: Adresa pri zapise do RAM:                510
# IB MODUL: Zapisovane data do RAM: 759489e2d9a8e612
# IB MODUL: Adresa pri zapise do RAM:                511
# IB MODUL: Zapisovane data do RAM: 5d750647bc9db552
# IB MODUL: Adresa pri zapise do RAM:                512
# IB MODUL: Zapisovane data do RAM: c217ce88d6178081
# IB MODUL: Adresa pri zapise do RAM:                513
# IB MODUL: Zapisovane data do RAM: 000000000000f436
```

Obr. 5.5: Výpis z IB modulu. Dáta transakcie sa zapisujú na konkrétne adresy v RAM.

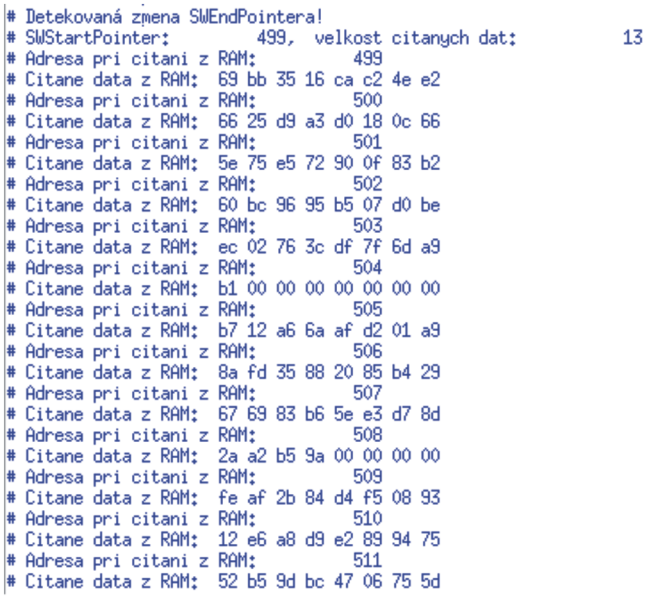
Krok 6. Výpis z monitora (obrázok 5.6) už zobrazuje chybnú transakciu. Znamená to, že dáta sa chybne vyčítali z RAM na základe SW ukazateľov. Ďalším postupom bude podrobná analýza diania v monitore.

```
# ----------------------------------------------------------
# -- MONITOR
# ----------------------------------------------------------
# Ifc: 0
# Packet no:          0, Packet size:         32, Data: feaf2b84d4f5089312e6a8d9e289947552b59dbc4706755d0000000000000000
# Packet no:          1, Packet size:          2, Data: 0000
```

Obr. 5.6: Výpis z monitora. Časť dát transakcie je chybných a majú hodnotu 0.

Krok 7. V monitore sa vykonávajú tri základné funkcie – detekcia zmeny SWEndPointera, načítanie dát z RAM na základe tejto zmeny a transformácia dát na FrameLinkovú transakciu, ktorá je následne odoslaná do scoreboardu. Z výpisu z monitora (obrázok 5.7) je zrejmé, že došlo k zmene SWEndPointera a táto zmena bola monitorom zachytená. Odštartuje sa načítavanie dát z adries v RAM, ktoré začínajú SWStartPointerom (hodnota 499) a končia SWEndPointerom (hodnota 511). Vidíme, že skúmaná transakcia začína na adrese 509. Problém je v tom, že čítanie dát končí adresou 511, zatiaľ čo dáta transakcie pokračujú na ďalšej stránke pamäte na adresách 512 a 513. Hodnota SWEndPointera je chybná, pretože sa porušuje celistvosť transakcie. Časť dát sa tak stratí, čo vysvetľuje nulové hodnoty v chýbajúcej časti transakcie.

Krok 8. V tejto chvíli sú k dispozícii všetky potrebné informácie. Treba kontaktovať autora komponenty, resp. systému a objasniť mu zdroj problému. On by mal chybu na základe týchto údajov opraviť, prípadne si sám spustí verifikačné prostredie a zorientuje sa v problematike.



Obr. 5.7: Detekcia zmeny SWEndPointera. Monitor prijíma dáta z adries v RAM, ktoré začínajú SWStartPointerom a končia SWEndPointerom.

5.1.4 Analýza výsledkov

Po oprave nájdených chýb je dôležité vyhodnotiť výsledky verifikácie. Treba zvážiť, do akej miery verifikácia spĺňa požiadavky (requirementy) na ňu kladené a akým spôsobom to dosahuje.

Zhrnutie requirementov pre RX DMA radič a spôsob ich overenia vo verifikácii:

1. Dodržanie verifikačných požiadaviek jednotlivých rozhraní.

   Spôsob overenia: Assertions na jednotlivých rozhraniach (analýza v ModelSime), implementácia podmienok v zdrojových súboroch.

2. Počet transakcií, ktoré sú do radiča odoslané driverom, je rovný počtu transakcií, ktoré sú z radiča prijaté monitorom.

   Spôsob overenia: Transakčná tabuľka v scoreboarde zostane po ukončení prenosu prázdna.

3. Transakcie sú do RAM ukladané v takom poradí, v akom sú do radiča zasielané driverom.

   Spôsob overenia: Porovnávanie transakcií v scoreboarde princípom FIFO.

4. Nemení sa dátový obsah prenášaných transakcií.

   Spôsob overenia: Porovnanie dátového obsahu transakcií v scoreboarde.

5. Transakcie zapisované do RAM nikdy nemajú cieľovú adresu mimo bloky definované v descriptoroch.

   Spôsob overenia: Keby mala transakcia pri zápise do RAM cieľovú adresu mimo bloky definované v descriptoroch, v transakčnej tabuľke v scoreboarde by nikdy nedošlo k zhode.

6. Dosiahnutie čo najväčšieho pokrytia.

   Spôsob overenia: Analýza coverage v ModelSime:

   Pokrytie kódu (code coverage) zdrojových súborov je možné vyhodnotiť na základe údajov získaných z ModelSimu. Po odstránení mŕtvych častí kódu je pokrytie 100%-né.

   Sledované funkčné pokrytie (functional coverage) je kompletné, ak sú otestované všetky konfigurácie (povolené kombinácie v nastavení generických parametrov), v rámci každej z nich všetky povolené rozsahy generovaných transakcií a nedochádza k žiadnym nezhodám v assertions. Pokrytie signálov (command coverage) v jednotlivých covergroups zobrazuje tabuľka 5.3, ktorá zhŕňa údaje získané z testovania v ModelSime (C.1). Vo väčšine prípadov je pokrytie 100%. Pokrytie na FrameLinkovom rozhraní však 100% dosiahne málokedy. Je to z toho dôvodu, že počas simulácie nie sú dosiahnuté všetky protokolom povolené kombinácie signálov.

Test	Transakcie	Coverage [%]													
		FrameLink				DMA Modul				DescManager				IB Modul	SW Modul
		0	1	2	3	0	1	2	3	0	1	2	3		
1.	5000	97	-	-	-	100	-	-	-	100	-	-	-	100	100
2.	5000	97	-	-	-	100	-	-	-	100	-	-	-	100	100
3.	2 x 5000	95	95	-	-	100	100	-	-	100	100	-	-	100	100
4.	2 x 5000	95	95	-	-	100	100	-	-	100	100	-	-	100	100
5.	4 x 4000	95	95	95	95	100	100	100	100	100	100	100	100	100	100
6.	4 x 4000	95	95	95	95	100	100	100	100	100	100	100	100	100	100

Tabuľka 5.3: RX DMA Command Coverage.

5.2 TX DMA radič

5.2.1 Implementácia – popis tried

1. **FrameLinkTransaction.** Táto trieda definuje formát FrameLinkových transakcií (paketov). Umožňuje napevno nastaviť počet častí paketu (ohraničené SOP, EOP) a rozsah, v ktorom sa bude dynamicky generovať veľkosť dát prenášaných v pakete. Dáta sa generujú náhodne. Obsahuje tiež metódy pre výpis (`display()`), kopírovanie (`copy()`) a porovnávanie (`compare()`) FrameLinkových transakcií.

2. **TxDmaCtrlDriver.** Každá instancia tejto triedy je pripojená na jedno FrameLinkové rozhranie a jeden mailbox. Mailbox posúva FrameLinkové transakcie driveru. Driver zasiela ich kópiu prostredníctvom callback funkcie do scoreboardu. FrameLinkové transakcie sú transformované na transakcie internej zbernice (`flTransactionToIbTransaction()`) a ukladané do RAM na adresu, na ktorú ukazuje SWEndPointer `putIntoRAM()`).

3. **TxSoftwareModul.** Obsahuje funkcie pre inicializáciu DMA radičov (`initCtrl()`), nastavenie SWEndPointera (`setEndPtr()`) a čítanie SWStartPointera (`getStrPtr()`) prostredníctvom rozhrania MI32.

4. **TxDescManager.** Pripravuje descriptory do fronty pre jednotlivé kanály, pričom aktuálny descriptor zasiela podľa potreby na príslušné rozhranie (`sendDescriptor()`). Akonáhle si DUT descriptor vyčíta (nastavením signálu DESC_READ), pripraví sa do fronty nový descriptor (`addDescriptor()`).

5. **TxDmaModul.** Prijíma radičom generované DMA požiadavky z bus-masterového rozhrania (`getDmaRequest()`) a vytvára z nich štruktúrovanú informáciu (`parseDmaRequest()`), ktorá obsahuje nasledujúce časti: globálna adresa do RAM, lokálna adresa do HW buffera, veľkosť prenášaných dát a tag. V takomto tvare je informácia uložená do mailboxu dmaMbx (`addRequest()`).

6. **TxIbusModul.** Vyberá rozčlenenú DMA požiadavku z mailboxu dmaMbx aby získal potrebné údaje na prenos medzi RAM a HW bufferom. Na základe informácie o veľkosti prenášaných dát vyčíta dáta z globálnej adresy v RAM a cez zápisové rozhranie internej zbernice (`sendTransaction()`) ich zasiela do DUT.

7. **FrameLinkMonitor.** Prijíma transakcie cez výstupné FrameLinkové rozhranie z DUT. Transformuje ich zo signálovej reprezentácie so objektovej a odosiela prostredníctvom callback funkcie do scoreboardu.

8. **Scoreboard.** Trieda zhromažďuje framelinkové transakcie zasielané callback funkciou z drivera (trieda ScoreboardDriverCbs) a z monitora (trieda ScoreboardMonitorCbs). ScoreboardDriverCbs ukladá transakcie do ScoreboardTable. ScoreboardMonitorCbs hľadá zhodu medzi prijatou transakciou a transakciou v ScoreboardTable a keď ju nájde, vymaže ju. Kvôli konkurentnému prístupu je implementovaný semafor.

9. **Coverage.** Trieda Coverage obsahuje niekoľko metód, ktoré agregujú informácie o pokrytí z rôznych coverage groups pre jednotlivé rozhrania TX DMA radiča. Získame tak prehľad o nastavení jednotlivých signálov a ich kombinácii počas simulácie.

5.2.2 Návrh testov, nastavenie generických parametrov

TX DMA radič som testovala prostredníctvom automatického generovania obsahovo náhodných FrameLinkových transakcií.

Pred testovaním je potrebné nastaviť generické parametre radiča (tabuľka 5.4):

Názov	Popis	Hodnoty
BUFFER_DATA_WIDTH	Dátová šírka TX buffera.	64
BUFFER_BLOCK_SIZE	Maximálny počet položiek v bloku TX buffera.	512
BUFFER_FLOWS	Počet kanálov.	1/2/4
BUFFER_TOTAL_FLOW_SIZE	Veľkosť TX buffera v bajtoch pre jeden kanál.	16384
CTRL_BUFFER_ADDR	Bázová adresa TX buffera.	0
CTRL_DMA_DATA_WIDTH	Dátová šírka DMA požiadavky.	8/16
PACKET_COUNT	Počet častí paketu.	2
PACKET_SIZE_MAX	Horná hranica pri generovaní veľkosti paketu.	1. paket 32, 2. paket 1530
PACKET_SIZE_MIN	Dolná hranica pri generovaní veľkosti paketu.	1. paket 0, 2. paket 1

Tabuľka 5.4: Generické parametre.

Na základe rôznych kombinácií generických parametrov som zostavila testy znázornené v tabuľke 5.5. Nastavenie kombinácií v ModelSime zobrazuje v prílohe obrázok B.2.

Generiky	Test 1		Test 2		Test 3		Test 4		Test 5		Test 6	
BUFFER_DATA_WIDTH	64		64		64		64		64		64	
BUFFER_BLOCK_SIZE	512		512		512		512		512		512	
BUFFER_FLOWS	1		1		2		2		4		4	
BUFFER_TOTAL_FLOW_SIZE	16384		16384		16384		16384		16384		16384	
CTRL_BUFFER_ADDR	0		0		0		0		0		0	
CTRL_DMA_DATA_WIDTH	8		16		8		16		8		16	
PACKET_COUNT	2		2		2		2		2		2	
PACKET_SIZE_MAX	32	1530	32	1530	32	1530	32	1530	32	1530	32	1530
PACKET_SIZE_MIN	0	1	0	1	0	1	0	1	0	1	0	1

Tabuľka 5.5: Návrh testov pre TX DMA radič.

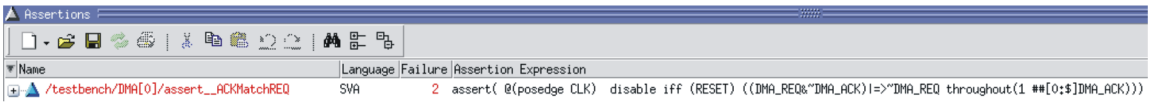
5.2.3 Analýza chybových stavov

Nasleduje analýza chyby v TX DMA radiči a postupnosť krokov pri jej hľadaní.

Krok 1. Počas simulácie nastane situácia, kedy stav na rozhraní DMA modulu nezodpovedá špecifikácii. K tomuto účelu boli pre každé z rozhraní vytvorené assertions, ktoré presne definujú nastavenie signálov v špecifických časových okamihoch. Na výstupe ModelSimu sa objaví chybové hlásenie (obrázok 5.8) a v okne Assertions (obrázok 5.9) môžeme zistiť, koľkokrát došlo k porušeniu protokolu rozhrania. Problém je ľahko identifikovateľný aj zo simulácie na obrázku 5.10. Vidno, že po vystavenom signáli DMA_REQ nepríde DMA_ACK, ale znovu DMA_REQ.

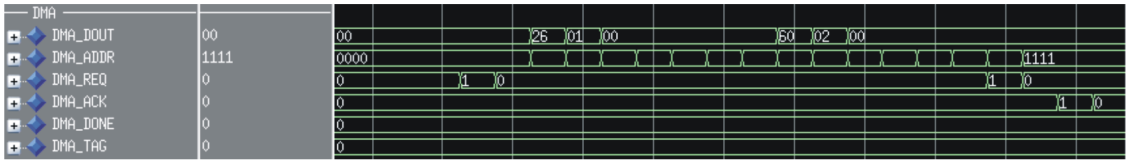
```
# ** Error: DMA_ACK is not active after DMA_REQ!
#    Time: 665 ns Started: 515 ns  Scope: testbench.DMA[0] File: ../../../../../comp/ver/interfaces/dma_ctrl_ifc.sv Line: 152
```

Obr. 5.8: Chybové hlásenie vyvolané porušením assertion bus-masterového rozhrania DMA modulu.



Obr. 5.9: Okno Assertions v ModelSime – výpis pre rozhranie DMA modulu.

Krok 2. Chyba spôsobí rozhodenie logiky radiča a na výstupné FrameLinkové rozhranie sa dostane neznáma transakcia (obrázok 5.11). Dôjde k nezhode v scoreboarde. Nie je splnená podmienka, že poradie transakcií, ktoré sú do radiča zasielané driverom, zodpovedá poradiu transakcií, ktoré sú prijaté monitorom. Transakcia, ktorá sa mala na výstupe radiča objaviť je zobrazená pod transakčnou tabuľkou.



Obr. 5.10: Chyba na rozhraní – signál DMA_REQ nie je potvrdený signálom DMA_ACK, nasleduje ďalší DMA_REQ.

```
# Unknown transaction received from monitor Monitor0
# Time: 1625,000 ns
#
# Interface: 0
# Packet no: 0, Packet size: 7, Data:
# 0000: 72 90 81 d8 e2 d5 16
# Packet no: 1, Packet size: 10, Data:
# 0000: 30 b2 0a 13 3b 71 2e c4  6f 16
# ------------------------------------------------------------
# --  TRANSACTION TABLE
# ------------------------------------------------------------
# Size:         100
# Items added:          100
# Items removed:            0
#
# Interface: 0
# Packet no: 0, Packet size: 12, Data:
# 0000: 08 a3 5b 53 5b d6 54 9b  50 99 01 2e
# Packet no: 1, Packet size: 19, Data:
# 0000: b9 5f 71 21 6a 67 3f be  58 d8 08 4f ec c5 1d 89
# 0010: 33 08 70
```

Obr. 5.11: Na vstup monitora prichádza neznáma transakcia.

5.2.4 Analýza výsledkov

Zhrnutie requirementov pre TX DMA radič a spôsob ich overenia vo verifikácii:

1. Dodržanie verifikačných požiadaviek jednotlivých rozhraní.

   Spôsob overenia: Assertions na jednotlivých rozhraniach (analýza v ModelSime), implementácia podmienok v zdrojových súboroch.

2. Počet transakcií, ktoré sú do RAM uložené driverom, je rovný počtu transakcií, ktoré sú z radiča prijaté monitorom.

   Spôsob overenia: Transakčná tabuľka v scoreboarde zostane po ukončení prenosu prázdna.

3. Transakcie sú z RAM vyberané v takom poradí, v akom sú do RAM zasielané driverom.

   Spôsob overenia: Porovnávanie transakcií v scoreboarde princípom FIFO.

4. Nemení sa dátový obsah prenášaných transakcií.

   Spôsob overenia: Porovnanie dátového obsahu transakcií v scoreboarde.

5. Transakcie čítané z RAM nikdy nemajú zdrojovú adresu mimo bloky definované v descriptoroch.

   Spôsob overenia: Keby mala transakcia pri čítaní z RAM zdrojovú adresu mimo bloky definované
   v descriptoroch, v transakčnej tabuľke v scoreboarde by nikdy nedošlo k zhode.

6. Dosiahnutie čo najväčšieho pokrytia.

   *Spôsob overenia:*Analýza coverage v ModelSime:

   Po odstránení mŕtvych častí kódu a zjednodušení podmienok je pokrytie kódu (code coverage) 100%-né. Tento stav bol dosiahnutý po analýze v ModelSime.

   Sledované funčné pokrytie (functional coverage) je kompletné, kedže boli otestované všetky konfigurácie, v rámci každej z nich všetky povolené rozsahy generovaných transakcií aj assertions. Pokrytie signálov (command coverage) v jednotlivých covergroups zobrazuje tabuľka 5.6, ktorá zhŕňa údaje získané z testovania v ModelSime (C.2). Vo vačšine prípadov má hodnotu 100%.

Test	**Transakcie**	**Coverage [%]**													
		FrameLink				**DMA Modul**				**DescManager**				**IB Modul**	**SW Modul**
		0	1	2	3	0	1	2	3	0	1	2	3		
1.	5000	95	-	-	-	100	-	-	-	100	-	-	-	100	100
2.	5000	95	-	-	-	100	-	-	-	100	-	-	-	100	100
3.	5000	95	95	-	-	100	100	-	-	100	100	-	-	100	100
4.	5000	95	95	-	-	100	100	-	-	100	100	-	-	100	100
5.	5000	94	94	94	94	100	100	100	100	100	100	100	100	100	100
6.	5000	94	94	94	94	100	100	100	100	100	100	100	100	100	100

Tabuľka 5.6: TX DMA Command Coverage

Kapitola 6

Záver

> „Everyone knows debugging is twice as hard as writing a program in the first place.“
>
> Brian Kernighan

Slová Briana Kernighana vyzdvihujú myšlienku, že samotná implementácia často nestačí. Celý projektový tím by si mal uvedomiť, že vo vyvíjanom systéme sú chyby. Verifikácia vo výraznej miere pomáha programátorom pri úsilí odhaliť ich a zneškodniť. Nie vždy ide o jednoduchú činnosť. Nájsť chyby býva niekedy časovo náročnejšie ako samotná implementácia.

Súčinnosť SystemVerilogu ako verifikačného jazyka a ModelSimu ako simulačného nástroja sa ukázala byť vhodnou pri odhaľovaní a lokalizácii chýb v návrhu.

Na základe teoretických znalostí o princípoch DMA prenosu, funkcionalite samotného hardwarového návrhu a verifikačných postupoch som v tejto bakalárskej práci navrhla a implementovala verifikačné prostredie DMA radičov. Ide o komplexný systém pozostávajúci s viacerých modulov, preto som uprednostnila testovanie automatickým generovaním transakcií. Ich počet musí byť dostatočne veľký, aby sa overili aj hraničné situácie ako zaplnenie bufferov, či prístup do rôznych častí pamäte. Pomocou assertions, ktoré som vytvorila na základe vopred stanovených verifikačných požiadaviek, sa mi podarilo odhaliť chyby súvisiace s porušením protokolov rozhraní. Pri testovaní je taktiež dôležité overiť kombinácie všetkých prípustných hodnôt generických parametrov radiča ako napr. počet vstupných a výstupných rozhraní, či veľkosť bufferov.

Verifikačné prostredie mi pomohlo odhaliť viaceré chyby, niektoré aj značne komplikované. Išlo najmä o chybné nastavenie softwarových ukazateľov pri zápise do rôznych stránok pamäte, chybné kombinácie signálov na rozhraniach a chyby v časovaní. Námet, ako postupovať po objavení chyby, som podrobne popísala v podkapitolách Analýza chybových stavov. Cieľom je získať dostatočné množstvo informácií pre autora hardwarového návrhu, ktorý tak jednoduchšie identifikuje vzniknutý problém a chybu čo najrýchlejšie opraví.

Akonáhle sa verifikácia dostane do stavu, kedy nedochádza k žiadnym nezhodám v scoreboarde (počet transakcií vstupujúcich do systému je rovný počtu transakcií z neho vystupujúcich), je možné vytvoriť analýzu výsledkov. Treba objasniť postup, akým boli overené jednotlivé verifikačné požiadavky a zhodnotiť celkové pokrytie. Keď pokrytie signálov na niektorom z rozhraní nedosahuje hodnôt blízkych k 100%, je nutné preskúmať situáciu kvôli potenciálnej chybe a zhodnotiť výsledok. Týmto je možné verifikáciu považovať za ukončenú.

Podľa môjho názoru je navrhnutý spôsob verifikácie hardwarového designu efektívny, prehľadný a rýchly. Osvedčil sa ako veľmi užitočný pomocník pri odhaľovaní zložitých chýb v projekte Liberouter, ktoré neboli inými prostriedkami odhalené a spôsobovali značné problémy.

Ohľadne budúceho vývoja verifikačných prostredí si myslím, že existujú spôsoby, ako verifikáciu ešte viac zefektívniť. V prvom rade by sa nemala podceňovať fáza návrhu, v ktorej je podstatná komunikácia s vedením projektu a autormi hardwarového návrhu. Treba včas definovať verifikačné požiadavky, ktoré treba v rámci verifikácie overiť a to nielen funkčné, ale aj tie, ktoré súvisia s dodržaním protokolu rozhraní použitých zberníc. Ak je to v danom projekte možné, navrhujem pri implementácii znovu používať už vytvorené komponenty, prípadne zvoliť dedičnosť pri rozšírení ich funkcionality. Pre potreby súčasných aj budúcich verifikátorov by sa mali návrhy prostredí, popis implementácie kľúčových komponent, ako aj postupy pri overovaní requirementov patrične dokumentovať. Čo sa týka nových postupov vo verifikácii, bolo by vhodné detailne preštudovať metodológiu OVM a inšpirovať sa komponentami vytvorenými v OVL.

Literatúra

[1] Wikipedie - Otevřená encyklopedie. `http://cs.wikipedia.org/wiki/DMA`.

[2] Cadence Design Systems, Mentor Graphics: *Open Verification Methodology User Guide*. 2008.

[3] Cohen, B.; Venkataramanan, S.; Kumari, A.: *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2004, iSBN 0-9705394-6-0.

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

[5] Šimková, D.: *Hardware pro začátečníky*. Grada, 2007, iSBN 80-247-2029-9.

[6] Kobiersky, P.; Malek, T.; Puš, V.: SystemVerilog Verification of VHDL Design. 2007.

[7] Kolektív autorov projektu Liberouter: Wiki projektu Liberouter. `https://www.liberouter.org`.

[8] Kotásek, Z.: Studijní opora k předmětu Periferní zařízení. `https://www.fit.vutbr.cz/study/courses/IPZ/public/`, 2008.

[9] Spear, C.: *SystemVerilog for Verification*. Springer, 2006, iSBN 0-387-27036-1.

Dodatok A

Verifikačné prostredie v ModelSime

ModelSim je simulačný nástroj, ktorý poskytuje prostredie pre verifikáciu v SystemVerilogu. Umožňuje analyzovať signály v simulácií, ako aj assertions a coverage. Nasledujúci príklad zobrazuje verifikačné prostredie RX DMA radiča pre 1 transakciu.

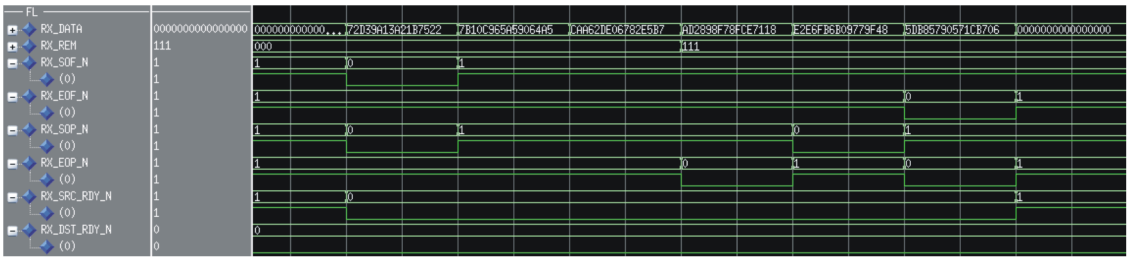
1. **Formát transakcie**

```
# Ifc: 0
# Packet no:          0, Packet size:          32, Data: 22751ba2139ad372a56490a565c9107bb7e58267e02da6ca1871ce8ff79828ad
# Packet no:          1, Packet size:          16, Data: 489f77096bfbe6e206b71c579057b85d
```

Obr. A.1: Transakcia

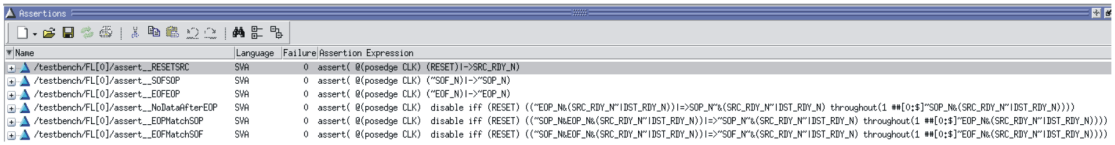
Formát transakcie je zrejmý z obrázka A.1. Transakcia bola zaslaná na rozhranie 0. Obsahuje dve časti, veľkosť prvej je 32 bajtov, veľkosť druhej 16 bajtov.

2. **FrameLink rozhranie**



Obr. A.2: FrameLink - rozhranie.

Na obrázku A.2 vidno prenos vyššie zobrazenej transakcie cez rozhranie FrameLink. Je zrejmé nastavenie signálu RX_SOF_N na začiatku dátového prenosu a signálu RX_EOF_N na konci dátového prenosu. Nastavenie signálov RX_SOP_N a RX_EOP_N určuje začiatok a koniec jednotlivých častí transakcie. Signál RX_SRC_RDY_N je aktívny počas celého prenosu dát.

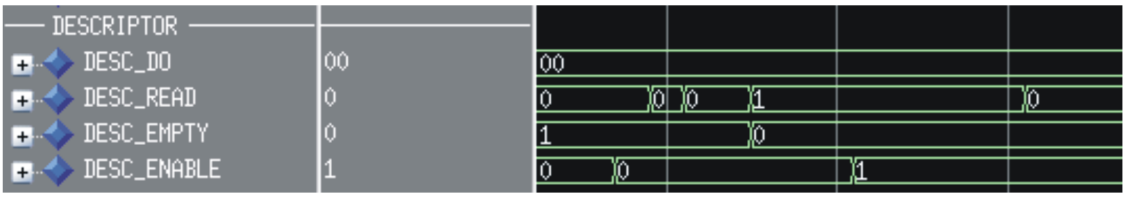


Obr. A.3: FrameLink - assertions.

3. **FrameLink assertions**

   Všetky FrameLinkové assertions (Assertion Expression) sú prehľadne zobrazené v okne ModelSimu (obrázok A.3). Stĺpec Failure zobrazuje, koľkokrát došlo k porušeniu jednotlivých assertions. Je zrejmé, že vo verifikácií RX DMA radiča sú všetky assertions splnené (Failure = 0).

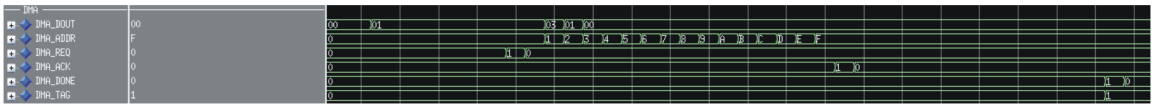
4. **Descriptor Manager rozhranie**



Obr. A.4: Descriptor Manager - rozhranie.

Podľa obrázka A.4 dochádza k prenosu descriptora s hodnotou 0 (DESC_DO). K prenosu dochádza pri nastavení hodnôt signálov rozhrania na nasledujúce hodnoty: DESC_READ = 1, DESC_EMPTY = 0, DESC_ENABLE = 1.

5. **DMA (Bus Master) rozhranie**



Obr. A.5: DMA - rozhranie.

Prenos 128-bitovej DMA požiadavky prebieha cez DMA bus-masterové rozhranie (obrázok A.5) postupne v niekoľkých taktoch. Akonáhle príde žiadosť (DMA_REQ = 1), uskutoční sa dátový prenos od nasledujúceho taktu hodinového signálu (DMA_DOUT je dátová hodnota požiadavky a DMA_ADDR je adresa prenášanej časti). Signál DMA_ACK potvrdzuje prijatie dát pripojenou komponentou a signál DMA_DONE spracovanie DMA požiadavky a ukončenie DMA prenosu.

Podľa obrázka je dátový obsah DMA požiadavky 00000000000000000000000000010301h.

Po rozčlenení je možné získať nasledujúce údaje:

- typ DMA požiadavky: 1h (prenos z hw do sw)
- veľkosť dát: 030h = 48 B (veľkosť je rovná veľkosti prenášanej transakcie)
- tag: 001h (číslo rozhrania*2+1)
- lokálna adresa: 00000000h
- globálna adresa: 0000000000000000h

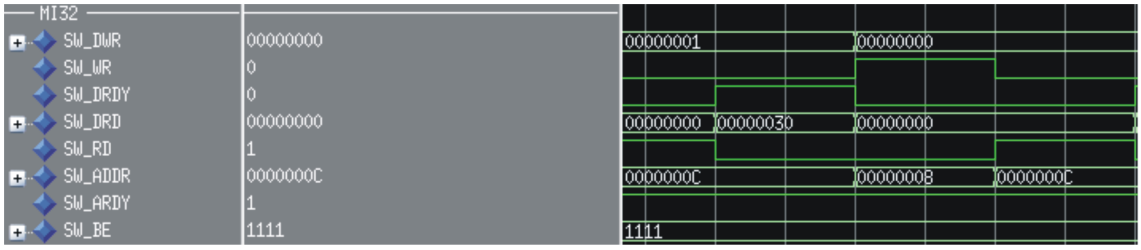
Všetky údaje sú správne a zodpovedajú špecifikácií.

6. **InternalBus rozhranie**

Obr. A.6: InternalBus - rozhranie.

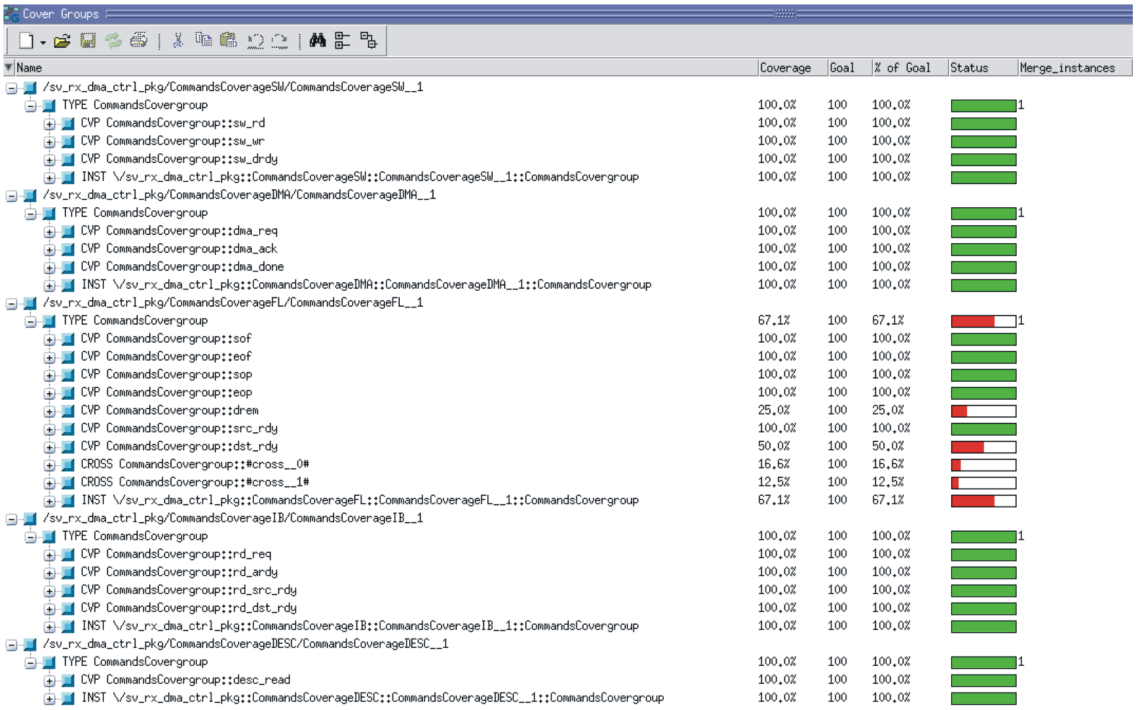
Prenos transakcie po internej zbernici je zrejmý z obrázka A.6. Žiadosť o čítanie z HW buffera je nastavená signálom RD_REQ a potvrdená signálom RD_ARDY. Zdroj potvrdí validitu dát pri nastavení signálu RD_SRC_RDY. Ak je cieľ v tomto okamihu pripravený (aktívny RD_DST_RDY), tak môže prijať dáta (RD_DATA) z lokálnej adresy v hw bufferi (RD_ADDR).

7. **MI32 rozhranie**



Obr. A.7: MI32 - rozhranie.

Cez rozhranie MI32 dochádza k čítaniu a nastaveniu SW ukazateľov. Na obrázku A.7 vidno nastavenie SWEndPointera. Je aktívny signál SW_WR, ktorý signalizuje požiadavku na zápis. Zapisované dáta sú dané obsahom signálu SW_DWR (ukazateľ má hodnotu 0) a zápis je realizovaný na lokálnu adresu, ktorú určí signál SW_ADDR.



Obr. A.8: Coverage.

8. **Coverage**

   ModelSim poskytuje prehľadné zobrazenie covergroups pre jednotlivé rozhrania (A.8). 100%-né pokrytie signálov vypovedá o tom, že počas simulácie boli nastavené všetky povolené hodnoty daného signálu. Na FrameLinkovom rozhraní sú vytvorené aj tzv. crosspoints, ktoré definujú povolené kombinácie rôznych signálov. Keďže v tomto prípade prešla rozhraním iba jedna transakcia, je jasné, že všetky povolené kombinácie nepokryla (coverage menej ako 20%).

Dodatok B

Generické parametre v ModelSime

```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 1
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  8
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    5000
#
```

```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 1
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  16
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    5000
#
```

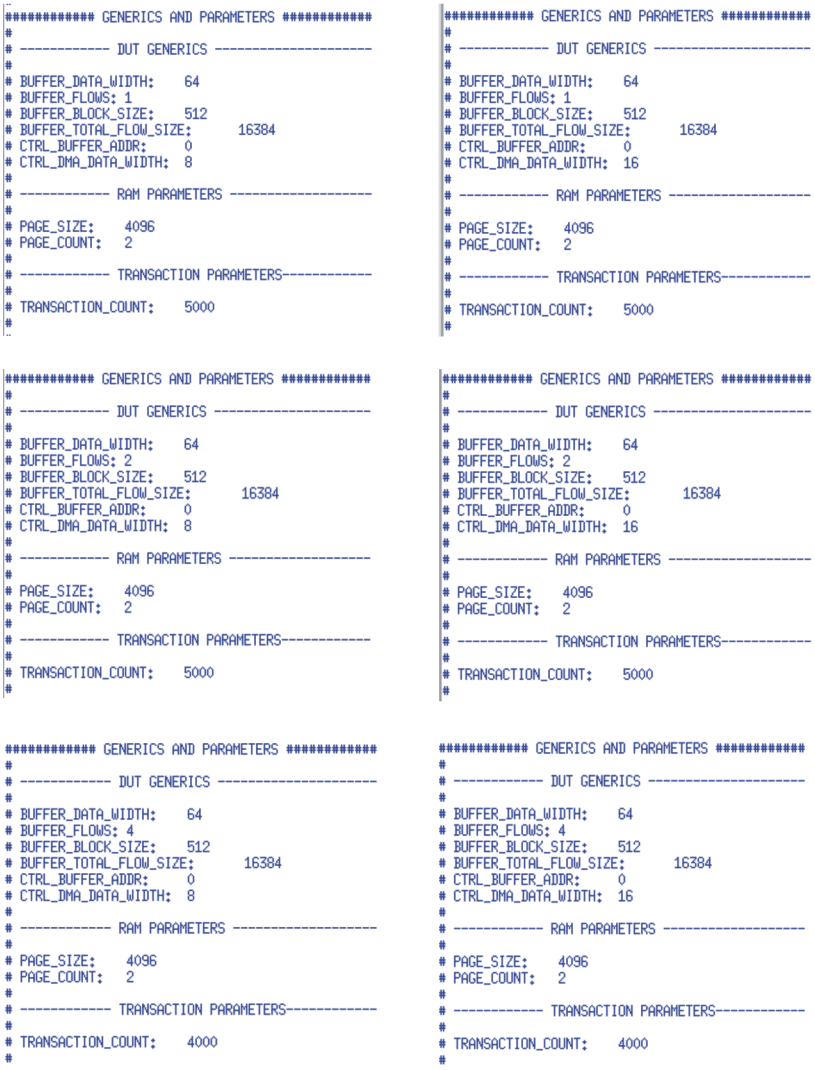
```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 2
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  8
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    5000
#
```

```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 2
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  16
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    5000
#
```

```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 4
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  8
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    4000
#
```

```
############ GENERICS AND PARAMETERS ############
#
# ------------ DUT GENERICS ----------------------
#
# BUFFER_DATA_WIDTH:    64
# BUFFER_FLOWS: 4
# BUFFER_BLOCK_SIZE:    512
# BUFFER_TOTAL_FLOW_SIZE:       8192
# CTRL_BUFFER_ADDR:     0
# CTRL_DMA_DATA_WIDTH:  16
#
# ------------ RAM PARAMETERS --------------------
#
# PAGE_SIZE:    4096
# PAGE_COUNT:   2
#
# ------------ TRANSACTION PARAMETERS------------
#
# TRANSACTION_COUNT:    4000
#
```

Obr. B.1: Nastavenie generických parametrov RX DMA radiča v ModelSime pri spustení testov.



Obr. B.2: Nastavenie generických parametrov TX DMA radiča v ModelSime pri spustení testov.

Dodatok C

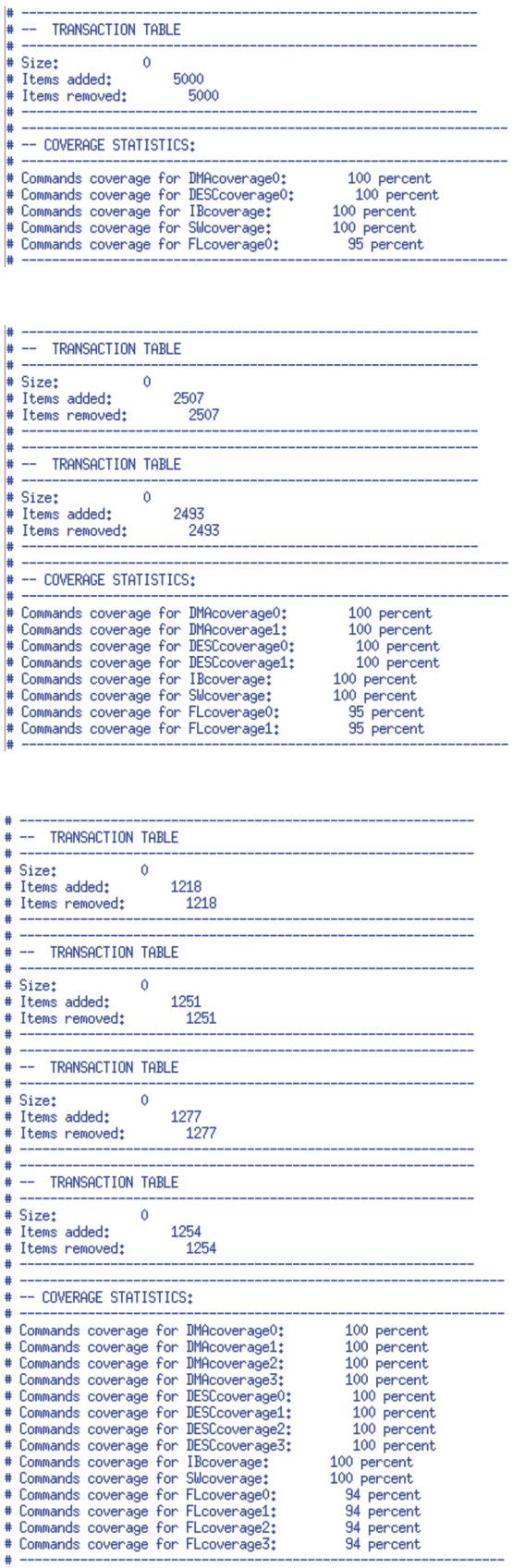
Výsledky testov v ModelSime

```
# ----------------------------------------------------------------
# --  TRANSACTION TABLE
# ----------------------------------------------------------------
# Size:          0
# Items added:        5000
# Items removed:      5000
# ----------------------------------------------------------------
# ----------------------------------------------------------------
# -- COVERAGE STATISTICS:
# ----------------------------------------------------------------
# Command coverage for FLcoverage0:         97 percent
# Command coverage for DMAModulcoverage0:      100 percent
# Command coverage for DESCcoverage0:       100 percent
# Command coverage for IBModulcoverage:       100 percent
# Command coverage for SWModulcoverage:       100 percent
# ----------------------------------------------------------------
```

```
# ----------------------------------------------------------------
# --  TRANSACTION TABLE
# ----------------------------------------------------------------
# Size:          0
# Items added:       10000
# Items removed:      10000
# ----------------------------------------------------------------
# ----------------------------------------------------------------
# -- COVERAGE STATISTICS:
# ----------------------------------------------------------------
# Command coverage for FLcoverage0:         95 percent
# Command coverage for FLcoverage1:         95 percent
# Command coverage for DMAModulcoverage0:       100 percent
# Command coverage for DMAModulcoverage1:       100 percent
# Command coverage for DESCcoverage0:       100 percent
# Command coverage for DESCcoverage1:       100 percent
# Command coverage for IBModulcoverage:       100 percent
# Command coverage for SWModulcoverage:       100 percent
# ----------------------------------------------------------------
```

```
# ----------------------------------------------------------------
# --  TRANSACTION TABLE
# ----------------------------------------------------------------
# Size:          0
# Items added:       16000
# Items removed:      16000
# ----------------------------------------------------------------
# ----------------------------------------------------------------
# -- COVERAGE STATISTICS:
# ----------------------------------------------------------------
# Command coverage for FLcoverage0:         95 percent
# Command coverage for FLcoverage1:         95 percent
# Command coverage for FLcoverage2:         95 percent
# Command coverage for FLcoverage3:         95 percent
# Command coverage for DMAModulcoverage0:       100 percent
# Command coverage for DMAModulcoverage1:       100 percent
# Command coverage for DMAModulcoverage2:       100 percent
# Command coverage for DMAModulcoverage3:       100 percent
# Command coverage for DESCcoverage0:       100 percent
# Command coverage for DESCcoverage1:       100 percent
# Command coverage for DESCcoverage2:       100 percent
# Command coverage for DESCcoverage3:       100 percent
# Command coverage for IBModulcoverage:       100 percent
# Command coverage for SWModulcoverage:       100 percent
# ----------------------------------------------------------------
```

Obr. C.1: Výsledky testov RX DMA radiča v ModelSime.



Obr. C.2: Výsledky testov TX DMA radiča v ModelSime.