VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

UVM VERIFIKAČNÍ PROSTŘEDÍ PRO SYSTÉM DMA MEDUSA

UVM VERIFICATION OF SYSTEM DMA MEDUSA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ZDENKO PETRUŠKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. LUKÁŠ KEKELY, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

144952

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Petruška Zdenko, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Vestavěné systémy
Název:	**UVM verifikační prostředí pro systém DMA Medusa**
Kategorie:	Vestavěné systémy
Akademický rok:	2022/23

Zadání:

1. Seznamte se se systémem DMA Medusa pro rychlý přenos dat mezi akcelerační kartou a operační pamětí hostitelského sytému, který je součástí NDK platformy.
2. Nastudujte základní principy metodiky UVM pro implementaci znovupoužitelných verifikačních prostředí a detaily současného verifikačního prostředí pro systém DMA Medusa.
3. Navrhněte implementaci verifikačního prostředí pro systém DMA Medusa podle metodiky UVM. Zachovejte testovací případy verifikované v současném verifikačním prostředí, případně navrhněte vhodná rozšíření. Zaměřte se přitom nejen na správnou funkčnost systému, ale také na ověření jeho výkonnostních parametrů a odhalení úzkých míst.
4. Implementujte navržené verifikační prostředí a vhodně ověřte jeho správnou funkčnost.
5. Porovnejte současné verifikační prostředí pro systém DMA Medusa s verifikačním prostředím podle metodiky UVM.
6. Diskutujte rozdíly mezi oběma variantami verifikačního prostředí a vliv těchto rozdílů na proces přípravy a realizace verifikace.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Kekely Lukáš, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	31.10.2022

Abstrakt

Práca pojednáva o návrhu a implementácií verifikačného prostredia systému DMA Medusa, ktorý je určený pre vysokorýchlostný prenos sieťových dát medzi pamäťou RAM a sieťovou kartou. Verifikačné prostredie je vytvorené podľa metodiky UVM. Jeho cieľom je odhaliť funkčné chyby pomocou náhodných testov. Pred implementáciou boli definované požiadavky na vytvorené prostredie. Požiadavky vychádzajú zo špecifikácie systému a analýzy predchádzajúceho prostredia, ktoré bolo implementované podľa odlišnej metodiky. Úlohou vytvoreného prostredia je implementovať funkcionalitu pôvodného, prípadne ju vhodne rozšíriť. Nové prostredie rozširuje generovanie stimulu v rámci pamäťového modelu. Naviac implementuje aj funkčné pokrytie vybraných vlastností. U pamäťového modelu je generovanie rozšírené o náhodné poradie odbavovania požiadavkov. Funkčným pokrytím sa overuje, že generovaný stimul spĺňa požadované vlastnosti. Zameriava sa na komunikáciu verifikovaného systému s pamäťou a sieťovou komponentou.

Abstract

This thesis describes design and implementation of verification environment for system DMA Medusa. DMA Medusa is hardware system used for high speed transmissions between network card and RAM. Verification environment is developed in SystemVerilog using UVM. Environment is designed with intention to find functional bugs using top level random stimulus. Testbench requirements have been defined prior to its implementation. Requirements are based on system specification and previous version of testbench. Previous version has been based on different methodology. New testbench implements the functionality of previous one. In addition, some functionality has been exteded. Implemented testbench extends previous memory model by serving memory requests in random order. It also implements functional coverage focused on communication with memory and network card. Goal of functional coverage is to monitor quality of generated stimulus.

Kľúčové slová

DMA, Funkčná verifikácia, UVM, SystemVerilog, Multi zbernice.

Keywords

DMA, Functional verification, UVM Methodology, SystemVerilog, Multi buses.

Citácia

PETRUŠKA, Zdenko. *UVM verifikační prostředí pro systém DMA Medusa.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Lukáš Kekely, Ph.D.

UVM verifikační prostředí pro systém DMA Medusa

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením doktora Lukáša Kekelyho. Uviedol som všetky literárne pramene, publikácie a dalšie zdroje, z ktorých som čerpal.

.....
Zdenko Petruška
17. mája 2023

Poďakovanie

Rád by som sa poďakoval vedúcemu práce doktorovi Lukášovi Kekelymu za odborné vedenie práce a jeho čas venovaný konzultáciam. Pri konzultáciach mi venoval vždy toľko času, koľko som potreboval. Tiež ma vždy vhodne nasmeroval k riešeniu konzultovaného problému. Rád by som sa poďakoval aj združeniu CESNET za umožnenie prístupu k systému DMA Medusa a možnosť odborných konzultácií v danej problematike. Pri konzultáciach mi predávali informácie nie len o verifikovanom systéme, ale často poradili, ako vyriešiť problémy pri návrhu a ako postupovať pri ladení vytvoreného prostredia.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Dnes sa stále zvyšujú nároky na výkon a priepustnosť počítačových sietí. V súvislostí s tým rastú aj požiadavky na rýchlosť spracovania sieťovej prevádzky. Pre pokročilé spracovanie sieťových dát v softvéri je potrebný ich prenos zo siete do operačnej pamäte počítača. Dáta sa musia prenášať s dostatočnou rýchlosťou. Za týmto účelom bol vytvorený systém *DMA Medusa*, ktorý slúži pre vysokorýchlostný prenos paketov medzi hardvérovými sieťovými komponentami a pamäťou *RAM*. Ide o číslicový systém určený pre platformu využívajúcu technológiu programovateľných hradlových polí (*FPGA*). Aby bolo možné zachovať dostatočnú priepustnosť, pre komunikáciu so sieťovými prvkami sú použité vysokorýchlostné multi zbernice a pre komunikáciu s pamäťou niekoľko koncových bodov. Tieto koncové body obsluhujú pamäťové požiadavky paralelne. Keďže prenesené pakety je potrebné aj efektívne spracovať v rámci softvéru, musia byť uložené na navzájom nezávislých miestach v pamäti tak, aby bolo umožnené paralelne spracovanie. Toto všetko vedie ku vzniku komplexného systému. Pre overenie jeho správnej funkčnosti by sa mali implementovať dostatočne kvalitné testy. Testy je potrebné vytvárať systematicky. Pri testovaní zložitých systémov pre *FPGA* sa dnes často používajú rovnaké techniky ako pri návrhu aplikačne špecifických integrovaných obvodov, teda funkčná a formálna verifikácia. U systémov pre technológiu *FPGA* sa používa najmä funkčná. Táto práca sa venuje vytvoreniu verifikačného prostredia pre systém *DMA Medusa*, ktoré vykonáva funkčnú verifikáciu.

Práca najskôr v kapitole 2 uvedie funkčnú verifikáciu a použitú metodiku. Kapitola 3 objasní multi zbernice, kapitola 4 základné princípy vysokorýchlostných prenosov sieťových dát a systém *DMA Medusa*. Následne je v kapitole 5 analyzovaná predošlá verifikácia systému. Po analýze sú definované požiadavky na nové prostredie kapitolou 6. Požiadavky vychádzajú zo špecifikácie systému, kontextu jeho použitia a analýzy predošlej verifikácie. Na základe požiadavkov je navrhnuté a implementované nové prostredie, ktorého architektúra je popísaná v kapitole 7. Prostredie využíva mechanizmy, ktoré vychádzajú z pôvodného prostredia a sú naviac rozšírené. Po vytvorení prostredia je definované, implementované a analyzované funkčné pokrytie generovaného stimulu. Kapitola 8 zhrnie funkčné pokrytie vrátane nepokrytých oblastí, ktoré boli vďaka nemu odhalené. Zhodnotenie vytvoreného prostredia je obsahom kapitoly 9.

Kapitola 2

Funkčná verifikácia a UVM

Verifikácia je proces, pri ktorom sa kontroluje, že chovanie verifikovaného systému vyhovuje chovaniu popísanému v množine jeho špecifikácií. Verifikácia číslicových obvodov je dnes veľmi rozšírená. Podľa štúdie [13] sa v projektoch pracujúcich na dizajne aplikačne špecifických integrovaných obvodov (*ASIC*) venuje viac inžinierov verifikácii, ako samotnému dizajnu. U projektov používajúcich technológiu programovateľných hradlových polí (*FPGA*) sa počet verifikačných inžinierov približuje k počtu dizajnérov. Pre verifikáciu číslicových obvodov sa dnes najviac používa funkčná a formálna verifikácia[2].

Formálna verifikácia[1] je typ statickej verifikácie, kedy sa matematicky dokazuje, že verifikovaný model má zhodné správanie s referenčným modelom pre všetky možné vstupy. Referenčný model je väčšinou definovaný pomocou množiny vlastností. Definované vlastnosti môžu zohrávať 2 úlohy pri verifikácii – môžu buď definovať vlastnosť samotného modelu, alebo definovať vlastnosť vstupu, na ktorú sa má brať ohľad pri verifikácii. Používané typy metód formálnej verifikácie sú:

- kontrola ekvivalencie (*Equivalence checking*),
- kontrola vlastností (*Property checking*).

Metódy pre kontrolu ekvivalencie porovnávajú dva modely popísané na rovnakej alebo odlišnej úrovni. Metódy možno ďalej deliť na kontrolu kombinačnej ekvivalencie (*Combinatorial equivalence checking*) a sekvenčnej ekvivalencie (*Sequential equivalence checking*). Metódy pre kontrolu kombinačnej ekvivalencie porovnávajú kombinačnú logiku medzi registrami, ktoré sa v modeloch nachádzajú. Kontrola Kombinačnej ekvivalencie sa používa napríklad pri porovnaní reprezentácie obvodu v rôznych stupňoch syntézy, pri pridaní testovacej logiky alebo pri aplikovaní rôznych optimalizácií. U metód kontroly sekvenčnej ekvivalencie sa kontroluje, či dva obvody reagujú na rovnakú sekvenciu vstupov rovnakou sekvenciou výstupov. Metóda sa používa pre porovnanie dvoch rôznych modelov na úrovni registrových prenosov (*RTL*), prípadne pre porovnanie reprezentácie jedného modelu na rôznych úrovniach. Využitie týchto metód môže mať veľký potenciál najmä pri syntéze obvodov popísaných na vyššej úrovni (*High level Synthesis*).

Metódy *Property checking*[1] tvoria ďalšiu oblasť metód formálnej verifikácie. Pri verifikácii sa nimi dokazuje, že je popísaná vlastnosť dodržaná pre každý vstup. Niekedy môže byť príliš náročné dokázať platnosť/neplatnosť danej vlastnosti, preto je možné dokázať platnosť/neplatnosť len ohraničene, typicky pre istý počet krokov, alebo obmedzením stavového priestoru verifikácie definovaním štartovacieho stavu. Prípadne možno stavový priestor obmedziť definovaním vlastností, o ktorých možno uvažovať, že v testovanom systéme vždy

platia. Článok [4] pojednáva o použití formálnej verifikácie ako doplnku k simulácií pri verifikácii procesoru. Formálnu verifikáciu navrhuje použiť pre:

1. verifikáciu menších modulov. Kvôli zložitosti formálnych nástrojov je tento prístup možné použiť len v obmedzene veľkých moduloch,

2. u väčších modulov pre pokrytie okrajových situácií, ktoré by inak boli zachytené simuláciou s veľmi malou pravdepodobnosťou. Formálna verifikácia by sa použila v tomto prípadoch ako doplnok k simulácií,

3. otestovanie oblastí, kde použitie simulácie nie je adekvátne.

Použitie verifikácie pomocou simulácie je preto nutné najmä pri verifikácii väčších modulov. V tejto práci sa zameriam na verifikáciu pomocou simulácie, konkrétne funkčnú verifikáciu. Formálnu verifikáciu by bolo možné použiť ako rozšírenie, pre overenie chybových stavov, prípadne pre overenie okrajových situácií.

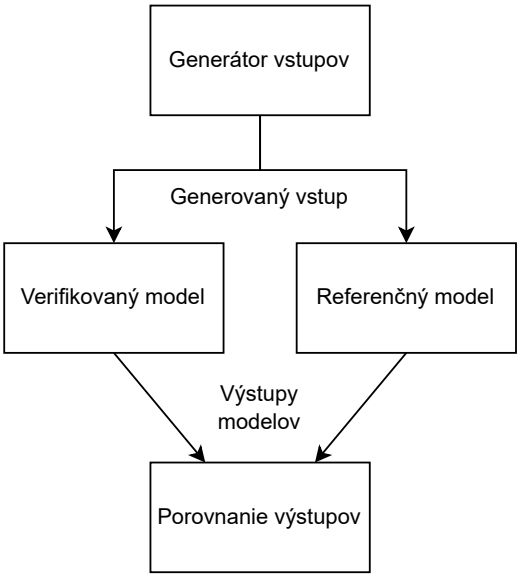
Funkčná verifikácia[2] je spôsob dynamickej verifikácie, kde sa porovnáva chovanie dvoch modelov verifikovaného systému. Jeden model je systém implementovaný hardvérovým dizajnérom, najčastejšie v jazyku *HDL* (*Hardware Design Language*), a druhý je referenčný model, typicky implementovaný verifikačným inžinierom. Dôležité je, aby oba tieto modely vznikli na základe špecifikácie, ideálne bez priameho kontaktu dizajnéra a verifikačného inžiniera. Takýmto spôsobom možno zamedziť situácií, kedy chovanie jedného z modelov bude odvodené z druhého modelu na základe znalostí, ktoré nie sú súčasťou špecifikácie.

Pre testovanie chovania modelov sa používajú simulačné alebo emulačné nástroje, prípadne prototypovanie (napríklad *FPGA* prototyp). Navrhovaný aj referenčný model sú testované pomocou rovnakých vstupov. Následne sa kontroluje, či sa výstup modelov zhoduje. Vstupy môžu byť pri tom priame, alebo náhodné. Náhodné vstupy sú synteticky generované, ich hodnota je typicky ovlyvnená obmedzeniami (*Random Constraints*). Takýto vstup sa nazýva stimul. Výstup z modelov pri danom stimule je porovnaný v komponente *Checker*. Komponenta *Checker* reportuje chybu v prípade rozdielneho výstupu medzi verifikovaným a referenčným modelom. Diagram 2.1 znázorňuje základný princíp funkčnej verifikácie. Generovaný stimul je vložený na vstup verifikovaného aj referenčného modelu, ich výstup je potom porovnaný. V prípade nezhody prostredie reportuje chybu.

Okrem použitia referenčného modelu je možné kontrolovať správnu funkčnosť verifikovaného systému aj sadou formálnych tvrdení. Formálne tvrdenie (*Assertion*) je výraz v temporálnej logike, ktorý musí vždy platiť. Tvrdenie možno použiť pri definovaní vlastností systému, reakcie na špecifický vstup, prípadne pri kontrole dodržiavania komunikačného protokolu. Tvrdenie možno zapísať pomocou jazykov zameraných na formálne tvrdenia. Najznámejšími sú jazyky *Property Specification Language* (*PSL*) a *SystemVerilog Assertions* (*SVA*).

Verifikovaný systém sa testuje náhodnými alebo priamymi vstupmi. Často môže byť náročné určiť, akým rôznym situáciám vystavujeme verifikovaný model aplikovaním náhodného stimulu. Platí, že funkčná verifikácia je len tak kvalitná, ako je kvalitný generovaný stimul. Pre získanie lepšieho prehľadu o použitom stimule možno použiť techniky pokrytia[5]. Merať možno pokrytie kódu verifikovaného systému alebo funkčné pokrytie. Simulačné nástroje podporujú oba typy pokrytia. U pokrytia kódu možno merať:

- pokrytie priradení (*Statement Coverage*),
- pokrytie vetiev (*Branch Coverage*),



Obr. 2.1: Základný mechanizmus funkčnej verifikácie.

- pokrytie podmienok (*Condition Coverage*),
- pokrytie výrazov (*Expression Coverage*).

Simulačné nástroje dokážu tiež merať pokrytie stavových automatov (*FSM Coverage*) a pokrytie prepínania logických hodnôt (*Toggle Coverage*). Funkčné pokrytie je typicky sada vlastností generovaného vstupu/výstupu, prípadne situácií, u ktorých chceme, aby nastali počas verifikácie. Túto sadu definuje verifikačný inžinier. Simulačný nastroj dokáže zbierať štatistiky o pokrytí takto definovaných bodov.

Verifikačné prostredie sa vytvára za účelom čo najlepšieho otestovania verifikovaného systému. Pred návrhom prostredia je potrebné určiť ciele, ktoré sa majú pri verifikácii dosiahnuť. Tieto ciele je vhodné spísať vo forme dokumentu, vznikne tak plán verifikácie. Plán verifikácie je kolekcia plánov testov. Testy sú definované za účelom pokryť konkrétnu časť verifikovaného systému. V rámci testovacieho plánu sa popisuje oblasť, ktorá sa má pokryť, definujú sa konkrétne body pokrytia a spôsob, akým sa dané body pokryjú. Na základe plánov testov sa vytvorí verifikačné prostredie, ktoré umožní implementovať potrebné testy.

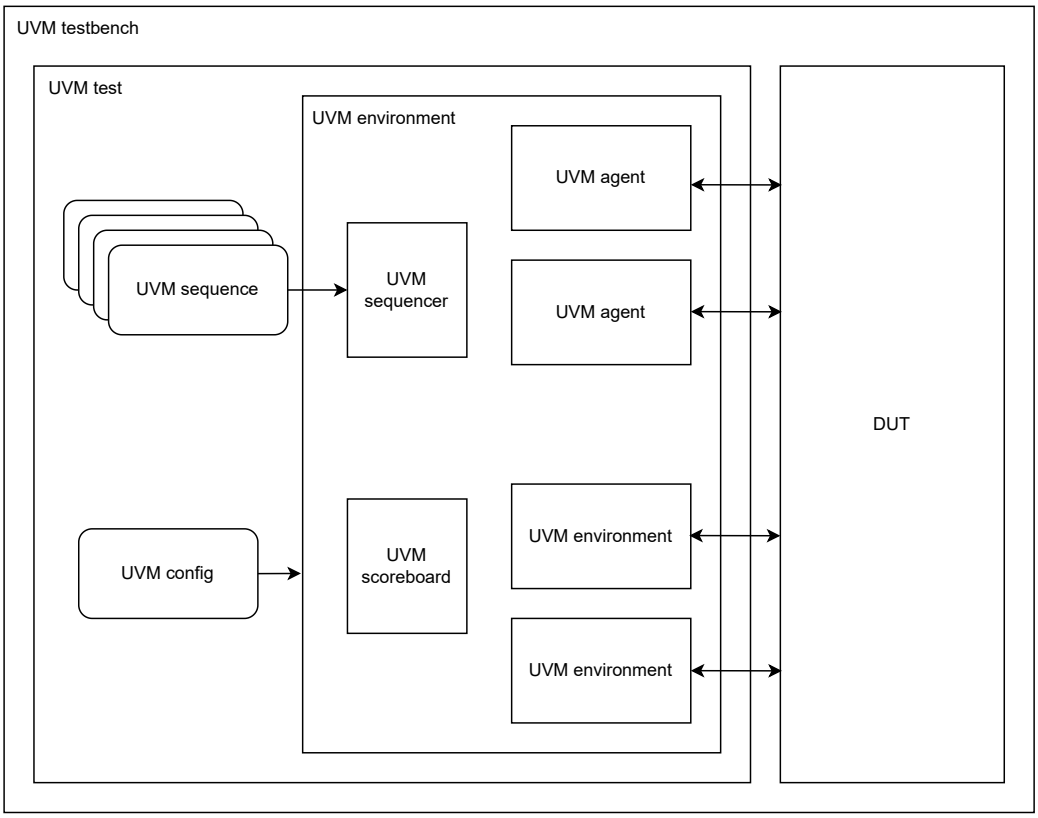
Prostredia možno vytvárať rôznym spôsobom. Dôležitá je okrem funkcionality aj prehľadnosť, rozšíriteľnosť, efektivita pri tvorbe a s tým súvisiaca aj možnosť znovupoužiteľnosti. Pri tvorbe prostredia možno použiť metodiku. Jednou z prvých metodík pre funkčnú verifikáciu bola metodika *e Reuse Methodology* (*eRM*). Bola vytvorená spolu s jazykom *e*, ktorý je jedným z prvých jazykov pre funkčnú verifikáciu (*HVL*). Metodika definovala konvenciu mien, rozdelenie prostredia na funkčné celky, sekvenciu a spôsob predávania správ. Na základe tejto tejto metodiky bola neskôr pripravená metodika *OVM* (*Open Verification Methodology*), ktorá už používala jazyk *SystemVerilog*. V jazyku využívala triedy, ktoré umožnili vyššiu mieru automatizácie tvorby.

Dnes je najpoužívanejšou metodika *UVM* (*Universal Verification Methodology*). Vychádza z hlavných princípov metodiky *OVM*. Popísaná bude podrobnejšie v nasledujúcej podkapitole.

2.1 Metodika UVM

Podľa štúdie [13] je metodika *UVM* (*Universal verification methodology*) dnes najrozšírenejšou metodikou v oblasti verifikácie číslicových obvodov pre technológie FPGA. Metodika je štandardizovaná [3]. Definuje súbor rozhraní pre programovanie aplikácií (*API*) a knižnicu základných verifikačných komponent *BCL* (*Base Component Library*), ktoré poskytujú základ pre vývoj modulárnych, škálovateľných a znovupoužiteľných prostredí. Pri správnom použití princípov daných metodikou možno výrazne urýchliť ich implementáciu. Metodika je podporovaná rozličnými simulačnými nástrojmi. Knižnica *BCL* je vytvorená pre jazyk *SystemVerilog*. Používa objektovo orientované programovanie. Metodika definuje, akým spôsobom možno riešiť často sa opakujúce problémy pri verifikácii. Konkrétne definuje princípy komunikácie, spôsob tvorby znovupoužiteľných modelov a prostredí, sadu základných tried pre verifikačné komponenty a systém reportovania. Používa pri tom pokročilé návrhové vzory, ako sú adaptéry, továrne metódy alebo abstraktná továreň.

Diagram 2.2 zobrazuje príklad testovacieho prostredia navrhnutého podľa metodiky *UVM* [12].



Obr. 2.2: Príklad testovacieho prostredia podľa metodiky *UVM*.

Hlavný modul *Testbench* typicky inštancuje modul *DUT* (*Device Under Test*) a triedu *Test*. Modul *DUT* zapuzdruje verifikovaný systém. Trieda *Test* je typicky inštancovaná za behu, aby bolo možné kompilovať verifikačné prostredie raz pre všetky definované testy. Trieda *Test* definuje prostredie *Environment*, konfiguruje toto prostredie a spustí špecifické sekvencie tak, aby sa čo najlepšie otestovala funkcionalita, na ktorú daný test cieli. Trieda

Environment (Prostredie) je hierarchická komponenta, ktorá spája dohromady verifikačné komponenty. Komponenty na najvyššej úrovni sú prepojené s *DUT*.

Trieda *Scoreboard* kontroluje vlastnosti *DUT*. Trieda pozoruje vstupné a výstupné transakcie verifikovaného systému. Výstupné transakcie porovnáva voči referenčným. Pre získanie referenčných výstupných transakcií typicky používa model, do ktorého vkladá vstupné transakcie. Model na základe vstupných transakcií predikuje výstup.

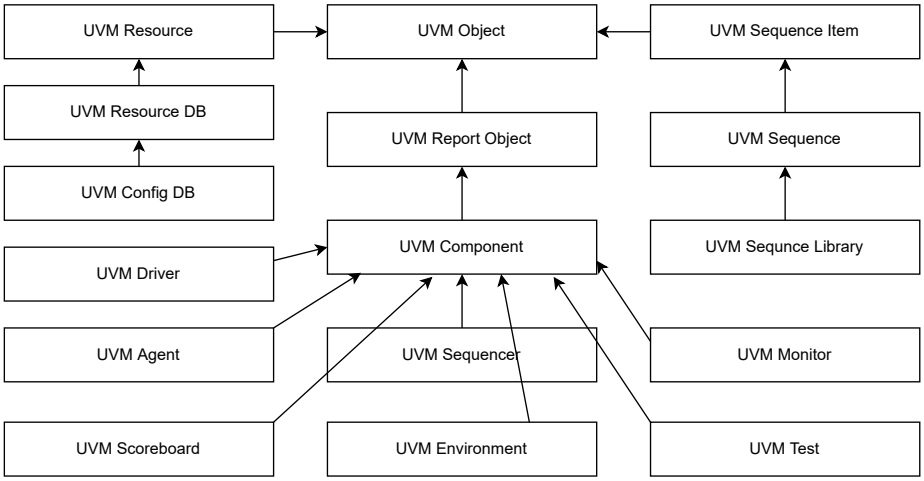
Trieda *Agent* je hierarchická komponenta. Táto komponenta typicky obsluhuje 1 rozhranie *DUT*. Komponenta obsahuje subkomponenty *Sequencer*, *Driver* a *Monitor*. Komponenta *Driver* v cykle získava transakcie z komponenty *Sequencer* a zapisuje ich na rozhranie verifikovaného modulu. Komponenta *Sequencer* vykonáva sekvenciu (*Sequence*) a rozkladá ju na jednotlivé položky sekvencie (*Sequence item*), ktoré potom predáva komponente *Driver*. Konkrétna sekvencia je na komponente *Sequencer* spustená nadradenou komponentou, väčšinou prostredím. Sekvencia definuje jednotlivé položky sekvencie, ktoré môžu tvoriť zaujímavý scenár pre otestovanie verifikovaného modulu. Monitor sleduje komunikáciu na rozhraní. Vyčítané transakcie prenáša do analytického portu (*TLM Analysis Port*), z ktorého môžu dané transakcie čítať analytické komponenty. Analytické komponenty môžu implementovať kontrolné mechanizmy (napríklad *Scoreboard*), prípadne funkčné pokrytie (*Functional Coverage*).

Dáta prenášané vo verifikačnom prostredí možno zabaliť do transakcie. Trieda *Transaction* definuje základnú triedu pre transakcie. V závislosti, v ktorej časti prostredia sú transakcie prenášané, môžu byť dáta v transakciách definované na rôznej úrovni abstrakcie. S klesajúcou úrovňou abstrakcie sa potom typicky pridáva informácia prenášaná v transakcií. V prípade, že sa transakcia používa pre generovanie dát, môže táto trieda obsahovať aj obmedzenia pre náhodné generovanie (*Constraints*). Takáto transakcia sa nazýva položka sekvencie.

Knižnica *BCL* (*UVM Base Component Library*) poskytuje hierarchické komponenty vrátane základných metód, ako sú metódy pre reportovanie a komunikáciu medzi komponentami, alebo továrne metódy, ktoré možno použiť napríklad pri definovaní knižnice sekvencií. Túto knižnicu možno potom použiť pri definovaní testu. Použitím hierarchických komponent a metód sa zvyšuje prehľadnosť kódu. Ich použitím sa tiež uľahčí správna implementácia podľa konceptov metodiky. Diagram 2.3 zobrazuje hierarchiu základných tried *UVM*. Trieda *Object* definuje základné metódy pre manipuláciu s objektom. Trieda *Component* implementuje fázovanie (*UVM Phases*). Každá trieda odvodená od tejto triedy môže následne definovať kód, ktorý sa bude vykonávať v jednotlivých fázach. Týmto princípom možno jednoducho definovať chovanie komponenty pri aktívnej hodnote *Reset*, prepojenie komponent, kód, ktorý sa bude vykonávať počas simulácie, prípadne kontrola a spôsob reportovania výsledkov po skončení simulácie.

Metodika používa základné princípy modelovania komunikácie na úrovni transakcií (*Transaction layer modeling*) [12]. Modelovanie na úrovni transakcií je dané štandardmi *TLM 1.0* a *TLM 2.0*. Tieto štandardy sa v metodike *UVM* používajú pre definovanie komunikačných rozhraní medzi komponentami. Pri komunikácii na úrovni transakcií sa prenášajú objekty *Transaction*. Komunikácia prebieha prostredníctvom portov. Základné triedy pre komunikáciu sú *TLM Port* a *TLM Export*. Trieda *TLM Port* definuje rozhranie pre programovanie aplikácií (*API*), trieda *TLM Export* potom implementuje dané rozhranie. Medzi portami možno komunikovať rozličnými spôsobmi – pomocou blokujúcej komunikácie, komunikácie prostredníctvom frontu, prípadne schránok (*Mailbox*).

Knižnica sekvencií [11] sa používa pre náhodné vytváranie a spúšťanie sekvencií. Základnou triedou pre knižnicu transakcií je trieda *Sequence Library*. Táto trieda je odvodená od



Obr. 2.3: Podstrom hierarchie tried *UVM*.

triedy *Sequence*. Hlavnou funkcionalitou knižnice sekvencií je náhodné vyberanie sekvencií zo zoznamu a ich spustenie. Nové sekvencie do knižnice možno pridať príslušnou metódou. Okrem toho je možné definovať princíp výberu sekvencie (náhodný, sekvenčný), minimálny a maximálny počet sekvencií. Knižnica sekvencií sa spúšťa na komponente *Sequencer* rovnako ako klasická sekvencia.

Vrstvenie sekvencií [11] sa používa pri protokoloch s hierarchickou štruktúrou. Takto možno generovať dáta nezávisle na protokole použitom pre ich prenos. Pri vrstvení sa používa vrstviaca komponenta a na najnižšej úrovni agent. Tento agent obsahuje komponentu *Driver*, ktorá zapisuje položky na najnižšej úrovni na rozhranie. Komponenta *Sequencer* tohto agenta typicky vykonáva prekladovú sekvenciu, ktorá zo sekvenčných položiek vyššej úrovne vytvára položky pre dané rozhranie. Sekvencie definujúce položky vyššej úrovne typicky bežia na komponentách *Sequencer* na vyššej úrovni, komunikujú pri tom so sekvenciou nižšej úrovne. Vrstvenie môže prebiehať aj pri monitorovaní rozhrania. Monitor na najnižšej úrovni číta transakcie na rozhraní. Transakcie následne posiela prostredníctvom analytického portu do monitoru vyššej úrovne. Tento monitor extrahuje z nízkoúrovňových transakcií transakcie vyššej úrovne, ktoré zašle analytickým komponentám.

Kapitola 3

Multi zbernice

Keďže systém *DMA* Medusa cieli na vysokú priepustnosť, často pri komunikácii používa multi zbernice. Porozumenie konceptu multi zberníc vrátane ich výkonnostných parametrov je vhodné najmä pre pochopenie rozsahu verifikácie systému, návrh verifikačného prostredia a pre definovanie zaujímavých situácií, ktoré je potrebné pokryť pri verifikácii. Kapitola uvedie motiváciu pre použitie týchto zberníc, spôsob prenosu rámcov dátovými slovami a porty, ktoré sa používajú v príslušných rozhraniach.

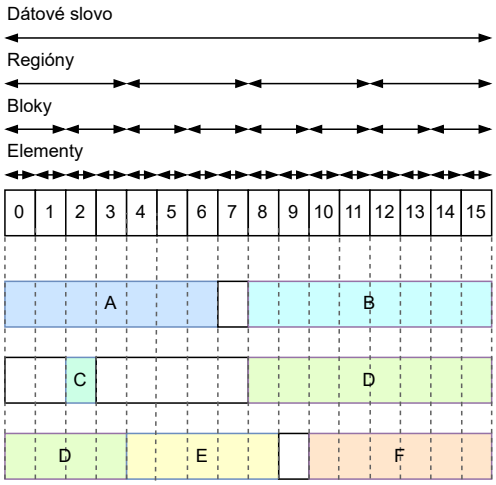
So zvyšujúcou priepustnosťou počítačových sietí sa zvyšujú aj nároky na priepustnosť zberníc, ktoré sieťové prvky používajú. Zvýšenie priepustnosti je možné zvýšením šírky zberníc. Dáta prenášané sieťami majú často variabilnú dĺžku. Pri použití zbernice, ktorá prenáša dáta zarovnane, môže nastať problém znižujúcej sa efektívnej priepustnosti. Ak by sme napríklad zbernicou širokou 512 bitov prenášali dáta o dĺžke 65 B, efektívna priepustnosť by sa znížila na 50% maximálnej priepustnosti. Typický ethernetový rámec má veľkosť 64 až 1518 bajtov. Efektívna priepustnosť by bola maximálna, ak by bol povolený nezarovnaný prístup, teda by sa v jednom prenesenom slove mohlo nachádzať niekoľko rámcov. V tomto prípade sa však zvýši náročnosť logiky, ktorá súvisí so zarovnaním. Zvýšil by sa pri tom počet použitých logických hradiel aj latencia prvkov, ako sú napríklad multiplexory. Podľa článku [8] môže spotreba počtu logických hradiel pri použití technológie *FPGA* rásť so zvyšujúcou šírkou exponenciálne. Článok preto definuje koncept multi zberníc, ktoré povoľujú prenos niekoľkých rámcov v jednom slove, pričom zavedie sadu pravidiel zarovnania. Sada pravidiel má za úlohu obmedziť réžiu spracovania.

Zbernica *Multi Frame Bus* (*MFB*) slúži pre jednosmerný prenos dátových rámcov medzi dvoma koncovými bodmi. Rámce môžu byť variabilnej dĺžky, zbernica povoľuje prenos viacerých rámcov v jednom slove. Rámec môže byť prenesený niekoľkými slovami. Jeden rámec sa prenáša spojito, bez medzier vnútri rámca. Rámce sa skladajú z elementov – najmenších rozlíšiteľných jednotiek (napríklad bajtov). Zbernica definuje pravidlá, ktoré sú kompromisom medzi maximálnou efektívnou priepustnosťou a réžiou spracovania. Prvým pravidlom je obmedzenie počtu rámcov, ktoré sa v slove môžu začať. Dátové slovo možno rozdeliť na regióny, kde v každom regióne môže začať maximálne jeden rámec a maximálne jeden skončiť. Tým dôjde k výraznému zníženiu zložitosti logiky, ktorá dáta spracuje. Medzi rámcami môžu byť vložené medzery. Druhým pravidlom je obmedzenie pozícií v rámci regiónu, kde sa nový rámec môže začať. Za týmto účelom sa regióny rozdelia na bloky. Rámec sa môže začať len na začiatku bloku. Tým dôjde k ďalšiemu zníženiu zložitosti.

Výsledná zbernica *MFB* sa skladá z nasledujúcich častí:

- región,
- blok,
- element.

Obrázok 3.1 ilustruje, ako sa dátové slovo delí na regióny a bloky. Na obrázku je zobrazená konfigurácia, kedy sa každý blok skladá z dvoch elementov, každý región z dvoch blokov a dátové slovo je rozdelené na štyri regióny. Na obrázku sa ďalej nachádzajú príklady, akým spôsobom možno prenášať dátové rámce jednotlivými slovami. Rámce musia vždy začať na začiatku bloku, kvôli čomu sa začiatok rámcov B a F musel posunúť o jeden element. V druhom dátovom slove sú ilustrované medzery, ktoré sa môžu ľubovolne pridávať medzi rámcami. Rámec C by mohol začať už na elemente 0. Podobne, rámec D by mohol začať už na elemente 4. Rámec D je prenesený v dvoch slovách. Rámce E a F ilustrujú legálne zdieľanie jedného regiónu.



Obr. 3.1: Spôsob prenosu dátových rámcov pri použití zbernice *Multi Frame Bus*.

Formálne možno zbernicu *Multi Frame Bus* popísať pomocou 4 parametrov:

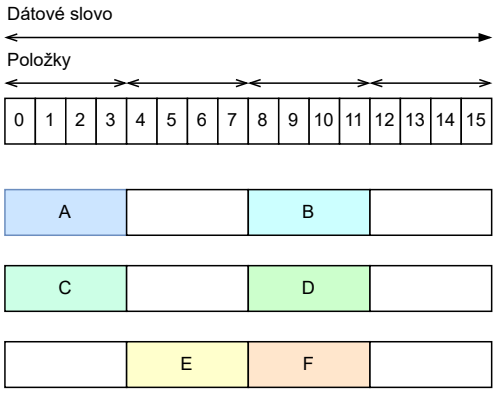
- počet regiónov (n),
- veľkosť regiónu (r) vyjadrený v počte blokov,
- veľkosť bloku (b) v počte elementov,
- veľkosť elementu v bitoch (e).

Pre zjednodušenie spracovania by veľkosti r a b mali byť obmedzené na mocniny 2. Šírka slova w sa týmito parametrami určí ako, $w = n \times r \times b \times e$. Článok [8] definuje skrátenú notáciu pre popis zbernice *Multi Frame Bus*, ako $MFB(n, r, b, e)$. Zbernicu ilustrovanú v 3.1 možno vyjadriť ako $MFB(4, 2, 2, *)$, kde $*$ vyjadruje akúkoľvek hodnotu.
Rozhranie zbernice *MFB* obsahuje nasledujúce porty:

- Dátové slovo (*Data*), jeho šírka je vyjadrená parametrom w.

- Príznaky začiatkov rámcov v slove (*SoF Flags*), ktoré definujú, v ktorých regiónoch sa začína nové slovo. Šírka portu je n.
- Vektor pozícií začiatkov rámcov (*SoF Positions*), ktoré definujú pozíciu bloku v rámci regiónu, kde sa rámec začína. Šírka portu je $n \times \lceil log_2(r) \rceil$.
- Príznaky koncov rámcov (*EoF Flags*) a vektor pozícií koncov (*EoF Positions*). Príznaky koncov rámcov označujú, či sa v danom regióne nejaký rámec končí obdobne ako v prípade príznakov začiatkov. Šírka portu je teda n. Rámec sa môže končiť nezarovnane ku začiatku bloku. Šírka portu *EoF Positions* je preto $n \times \lceil log_2(r \times b) \rceil$.
- Porty pre ovládanie toku. Tok môže byť ovládaný jednosmerne prostredníctvom portu *Source Ready*, alebo obojsmerne, pridaním portu *Destination Ready*.

K prenášaným dátovým rámcom je často potrebné preniesť aj príslušné metadáta. Metadáta sú typicky položky s pevne danou šírkou. Článok [8] definuje zbernicu *Multi Value Bus*, ktorá bola navrhnutá pre prenos metadát k rámcom prenášaným na zbernici *Multi Frame Bus*. Dvojica zberníc *MFB* a *MVB* tak často spolu tvoria jeden komunikačný kanál. Zbernica dokáže preniesť niekoľko položiek za takt. Obrázok 3.2 ilustruje, akým spôsobom možno metadáta prenášať v jednotlivých slovách. Na obrázku je zobrazená konfigurácia so štyrmi položkami. Na obrázku možno vidieť prenesené položky $A - F$. Tieto položky sú zarovnané na rovnaké pozície ako pozície začiatkov príslušných rámcov v obrázku 3.1. Všeobecne však neplatí žiadne pravidlo pre fixovanie pozícií rámcov.



Obr. 3.2: Štruktúra slova pri použití zbernice *Multi Value Bus*.

Obdobne ako v prípade zbernice *MFB*, článok [8] definuje skrátenú notáciu pre vyjadrenie konfigurácie - $MVB(m, i)$. Parametre m a i značia:

- počet položiek (m),
- šírku položky (i) uvedenú v počte bitov.

Pomocou týchto atribútov sa šírka slova vyjadrí ako $w = m \times i$. Konfiguráciu na obrázku 3.2 možno vyjadriť ako $MVB(4, *)$, kde symbol $*$ zastupuje ľubovolnú hodnotu.
Rozhranie zbernice *MVB* sa bude skladať z nasledujúcich portov:

- Dátové slovo (*Data*), ktoré prenáša položky jedného slova. Šírka portu je w.
- Príznaky signalizujúce validné položky (*Valid Flags*). Šírka portu je m.

- Porty pre ovládanie toku podobne, ako u *MFB*.

Metadáta môžu obsahovať niekoľko polí. Polia môžu byť prenášané štruktúrovane alebo serializovane. V prípade serializovanej podoby sa všetky polia skonkatenujú do jednej hodnoty a prenesú sa jedným portom. V prípade štruktúrnej podoby sa jednotlivé polia prenášajú pomocou separátnych portov. Serializovaná podoba je vhodná najmä pre všeobecné moduly, ako sú zreťazené linky alebo fronty *FIFO*. Štruktúrna podoba je vhodná najmä pre funkčné moduly, ktoré obsahujú logiku pre spracovanie polí.

Metadáta možno spolu s príslušnými rámcami prenášať aj priamo zbernicou *MFB*. Stačí pri tom pridať dátové slovo, ktoré bude obsahovať položky metadát. Metadáta sa potom môžu prenášať zarovnane s regiónom, kde príslušný rámec začína alebo končí. Dátové slovo pre metadáta bude mať teda rovnaký počet položiek, ako je počet regiónov. Validné položky budú v prípade zarovnania metadát so začiatkom rámcov signalizované portom *SoF Flags*. V prípade zarovnania s koncom rámcov možno použiť port *EoF Flags*.

Kapitola 4

Systém DMA Medusa

Sieťové pakety možno medzi aplikáciou a sieťovým rozhraním prenášať rôznym spôsobom. Najjednoduchším je prenos jednotlivých paketov pomocou štandardného sieťového zásobníku (*Network Stack*), ktorý je priamo súčasťou operačného systému. Spracovanie jednotlivých jeho sieťových vrstiev je v softvéri výpočetne náročné. Preto aplikácie, ktoré sú kritické na priepustnosť, často obchádzajú sieťový zásobník. Pri takýchto aplikáciach je ďalej snaha minimalizovať kopírovanie prenášaných dát v pamäti a maximalizovať úroveň paralelizácie ich spracovania.

Priamy prístup do pamäte (*Direct memory access*, skrátene *DMA*) je princíp priameho prenosu dát medzi hardvérovým podsystémom a hlavnou pamäťou počítačového systému, typicky pamäťou s náhodným prístupom (*RAM*). Procesor pri použití *DMA* najskôr inicializuje prenos poskytnutím inštrukcií o tom, ako sa má prenos vykonať. Následne je prenos riadený hardvérovým podsystémom nezávisle na práci centrálnej výpočetnej jednotky (*CPU*). Použitím priameho prístupu do pamäťe možno tak potenciálne zvyšiť výpočetný výkon procesoru, ktorý nemusí proces prenosu riadiť priamo. Zvyšuje sa tiež priepustnosť, pretože rýchlosť prenosu nie je limitovaná rýchlosťou vykonávania inštrukcií procesoru. Hardvérovým podsystémom môže byť napríklad sieťová komponenta. Úlohou systému *DMA Medusa* je vysokorýchlostný prenos paketových dát medzi sieťovou komponentou a hostiteľskou pamäťou. V tejto kapitole je popísaný systém *DMA Medusa*. Systém bol vytvorený pre použitie na platforme *FPGA*. V kapitole sú najskôr vysvetlené vysokorýchlostné paketové prenosy medzi pamäťou a sieťovou kartou. Následne sú predstavené konkrétne koncepty použité v systéme *DMA Medusa* a záver kapitoly sa venuje jeho základnej štruktúre. Pri popise štruktúry je kladený dôraz na vonkajšie rozhrania a komunikáciu prostredníctvom rozhraní. Tieto informácie sú dôležité pre definovanie požiadavkov na verifikačné prostredie a jeho architektúru.

4.1 Vysokorýchlostné pamäťové prenosy

Ako už bolo spomínané, pre maximálne využitie prenosovej rýchlosti je vhodné, aby neboli pakety v softvéri prenášané pomocou sieťového zásobníku. Aplikácia spracujúca dáta by pre minimalizáciu réžie mala ideálne pristupovať priamo do adresového priestoru, kde sú uložené dáta na prenos. Systémy *NDP* alebo *DPDK* umožňujú takýto prístup. Systém *XDP* umožňuje niektoré pakety spracovať priamo v pamäťovom priestore používanom pre prenos a niektoré zase v aplikácii.

Prenášané pakety majú variabilnú dĺžku. V prípade prenášania ethernetových rámcov je minimálna dĺžka rámca 64 B, pričom sa rámec skladá z 18 B hlavičky a 46 B dát. Maximálna dĺžka rámca je 1518 B (18 B pre hlavičku a 1500 B pre dáta). Na prenosovú rýchlosť má veľký vplyv spôsob, ktorým sú prenášané pakety uložené v hostiteľskej pamäti. Systémy pre vysokorýchlostné prenosy sieťových dát možno podľa spôsobu ich uloženia deliť na prúdovo alebo paketovo orientované. Prúdovo orientované prenosy majú v pamäti virtuálne spojitú vyrovnávaciu pamäť, do ktorej aplikácia zapisuje/číta pakety. Takýto prístup je veľmi výhodný pre hardvér, pretože umožňuje prenášať pakety bez ohľadu na zarovnanie. Dátové bloky možno tiež prenášať paralelne. Nevýhodou je však spracovanie paketov softvérom. Aby mohol softvér zapísať/prečítať paket, musí poznať, kde končí predošlý. To je možné určiť z počiatočnej adresy a dĺžky predošlého paketu. Pri paralelnom spracovaní paketov na strane softvéru takýto prístup vytvára potenciálne lineárnu závislosť medzi nimi. Predstaviteľom systému, ktorý používa prúdové prenosy paketov, je systém *NDP* (*Netcope Data Plane*). Systém *NDP* poskytuje knižnicu umožňujúcu zdieľanie pamäťového priestoru medzi ovládačom a aplikáciami, ktoré prijímajú/odosielajú dáta. Pakety sú v pamäťovom priestore uložené za sebou, s maximálnym zarovnaním 8 B. Dĺžka paketu je uložená v prvých 2 bajtoch paketu. Problémom pri paralelnom spracovaní viacerými aplikáciami je tiež koherencia vyrovnávacích pamätí (*Cache Coherency*), ktorá môže prístupy do pamäťového priestoru serializovať.

U paketovo orientovaných prenosov sú pakety uložené na rôznych miestach v pamäti. Toto prináša nevýhodu z pohľadu hardvéru, pretože kvôli variabilnej veľkosti paketov sa môže znížiť priepustnosť. Tento prístup však umožňuje lepšie paralelne spracovanie softvérom. Na princípe paketovo orientovaných prenosov pracujú systémy *DPDK* a *XDP*. Systém *DPDK* (*Data Plane Development Kit*) obsahuje sadu knižníc, ktoré podobne ako v prípade systému *NDK* umožňujú ovládaču a užívateľskej aplikácií pracovať s rovnakým pamäťovým priestorom. Paket môže byť tiež rozdelený na viac častí, systém *DPDK* prenáša pakety zarovnané na 4 B. Systém *XDP* (*Express Data Path*) umožňuje prístup k paketom na najnižšej úrovni prostredníctvom jadra operačného systému. Systém umožňuje predložiť jadru systému jednoduchú kompilovanú aplikáciu, ktorá prijatý paket vyhodnotí a rozhodne, ako paket ďalej spracovať. Jednou z možností je preposlanie paketu naspäť do sieťovej komponenty. Systém *XDP* umožňuje prijatý paket odoslať naspäť do siete bez nutnosti kopírovania. Výhodou systému je tiež možnosť priameho napojenia na virtuálny stroj. Systém vyžaduje, aby prijaté pakety boli uložené na rozdielnych stránkach. Dovoľuje tiež rozložiť paket na viac miest v pamäti.

4.2 Princípy použité v systéme DMA Medusa

Systém *DMA Medusa* [9] umožňuje prenášanie paketových dát smerom z hostiteľskej pamäte do sieťovej komponenty (smer *TX*) a opačne, zo sieťovej komponenty do hostiteľskej pamäte (smer *RX*). V smere *TX* tento systém dostane od hostiteľského systému informácie o paketoch, ktoré sú uložené v pamäti. Systém tieto pakety prečíta z hostiteľskej pamäte a zapíše ich na výstupné rozhranie, ktoré je pripojené k sieťovej komponente. V smere *RX* je úlohou systému prečítať pakety na vstupnom rozhraní, ktoré je napojené na sieťovú komponentu. Prijímanie paketov je inicializované hostiteľským systémom zo softvéru, úlohou hostiteľského systému je alokovať v hlavnej pamäti miesto pre uloženie paketov. Hostiteľský systém potom predá informácie o alokovanom mieste systému *DMA* a prenos inicializuje. Po prijatí informácií začne systém *DMA* zapisovať prijaté pakety do hostiteľskej pamäte. Systém *DMA* podporuje prenos paketov s minimálnou dĺžkou bez cyklického redundant-

ného súčtu (*CRC*), teda minimálna podporovaná veľkosť je 60 *B*. Podporovaný je aj prenos *Jumbo* paketov, ktoré vznikajú spájaním menších. Maximálna veľkosť výsledného paketu je 16 *KB*.

Systém *DMA* a hostiteľská pamäť spolu komunikujú prostredníctvom zbernice *PCIe* (*Peripheral Component Interconnect Express*) generácie 3 alebo 4 [9]. Zbernica *PCIe* umožňuje komunikáciu pomocou nezávislých paralelných kanálov. Pri použití 16 paralelných kanálov zbernice *PCIe Gen 3* možno dosiahnuť priepustnosť celkovo až 100 Gb/s, alebo až 200 Gb/s pri prenose zbernicou *PCIe Gen4*. Maximálnu priepustnosť nie je možné plne využiť za každých podmienok. Zbernica *PCIe* obmedzuje prenos dát definovaním maximálnej veľkosti zápisu/čítania jedným požiadavkom. Maximálna veľkosť pri zápise je daná parametrom *MPS* (*Maximum Payload Size*), pri čítaní parametrom *MRRS* (*Maximum Read Request Size*). Ďalej platí, že v prípade požiadavku na čítanie nemôže jeden požiadavok čítať dáta z viac stránok pamäte. Tieto parametre komunikácie obmedzujú maximálnu priepustnosť, konkrétne v prípade požiadavkov, ktorých veľkosť nie je zarovnaná na hodnoty *MRRS* a *MPS*, alebo v prípade prístupu, ktorý nie je zarovnaný na stránky. Zbernica *PCIe* dosahuje vysoké priepustnosti aj vďaka tomu, že požiadavky na čítanie sa odbavujú mimo poradia. Jednotlivé požiadavky sú následne párované na základe rovnakej hodnoty *Tag*. Hodnota *Tag* je nastavená v požiadavku a použitá hodnota musí byť v danom čase jedinečná. Systém *DMA* podporuje odbavovanie požiadavkov mimo poradia. Platí tiež, že prečítané dáta môžu byť rozdelené na niekoľko častí. Časti jednej odpovede prichádzajú v poradí. Posledná časť odpovede je označená príznakom *Last*.

Systém *DMA* Medusa podporuje paketovo orientované prenosy. Pre definovanie komunikácie medzi softvérovým ovládačom a hardvérovým modulom *DMA* bol vytvorený protokol *NPP* (*Netcope Packet Plane*). Protokol bol vytvorený predovšetkým pre podporu systému *XDP*. Systém *DMA* a ovládač spolu komunikujú prostredníctvom konfiguračných registrov. Registre sú uložené v rámci *DMA* modulu a ovládač k nim pristupuje prostredníctvom čítacích/zápisových požiadavkov. Protokol používa pre prenosy paketov pamäťové deskriptory. V smere *TX* tieto deskriptory popisujú pamäť, kde sú uložené pakety pre odoslanie. V smere *RX* deskriptory popisujú pamäťové bloky, kam sa prijaté pakety majú uložiť. Pri vysokorýchlostných prenosoch je dôležitý rýchly prenos deskriptorov. Napríklad pri rýchlosti 100 Gb/s a pri priemernej veľkosti paketu 512 B je potrebné preniesť a spracovať až 25 miliónov deskriptorov za sekundu. Z tohto dôvodu sú deskriptory navrhnuté tak, aby bola veľkosť jedného deskriptoru 64 bitov. V smere *TX* potrebuje deskriptor preniesť:

- 64 bitovú adresu začiatku paketu,
- dĺžku paketu,
- informáciu, či ide o posledný deskriptor obsahujúci dáta daného paketu,
- metadáta, ktoré budú predané sieťovej komponente spolu s paketom.

Keďže sa tieto informácie nezmestia do jedného deskriptoru, je zavedené stavové riadenie pri ich spracovaní – ak daná položka nie je uvedená v aktuálnom deskriptore, platí hodnota položky z posledného deskriptoru, ktorý ju obsahoval. Adresa pamäťového bloku s paketom sa prenáša rozdelená na horných 34 a spodných 30 bitov. Deskriptor bloku prenáša spodných 30 bitov, 16 bitovú veľkosť bloku (paketu) a metadáta. Horných 34 bitov sa prenáša konfiguračným deskriptorom. Ak dva po sebe nasledujúce pakety zdieľajú horných 34 bitov adresy, konfiguračný deskriptor nie je potrebné vytvárať a prenášať opakovane.

V smere *RX* potrebuje deskriptor preniesť:

- 64 bitovú adresu začiatku bloku,
- dĺžku alokovaného bloku,
- informáciu, či v prípade, že bude paket dlhší ako veľkosť bloku, možno zapísať zvyšok paketu do pamäťového bloku popísaného nasledujúcim deskriptorom.

Podobne, ako v smere *TX*, aj v smere *RX* sa adresa rozdelí na horných 34 a spodných 30 bitov, pri čom horných 34 sa prenáša konfiguračným deskriptorom. Ostatné informácie sa prenášajú deskriptorom bloku.

Deskriptory [7] sú uložené v jednom spojitom bloku pamäte, aby mohol modul *DMA* prečítať niekoľko deskriptorov naraz. Inak by réžia spojená s ich prenosmi mohla mať negatívny vplyv na priepustnosť zbernice. Blok pamäte by mal byť dostatočne veľký, ideálne s veľkosťou *MRRS*. K pamäťovému bloku sa pristupuje ako ku kruhovej vyrovnávacej pamäti. Táto kruhová pamäť je definovaná počiatočnou adresou a maskou, ktorá určuje jej veľkosť. Veľkosť pamäte je spravidla mocninou 2. Na validné položky odkazujú dve hodnoty, softvérový a hardvérový ukazateľ. Pri začatí prenosu sú oba ukazatele vynulované. Softvérový ukazateľ je aktualizovaný ovládačom, ktorý alokuje pamäť a chystá nové deskriptory. Hodnota aktuálneho softvérového ukazateľa je uložená v registri *DMA* modulu. Ovládač ho aktualizuje zápisom do registra. Keď *DMA* modul použije daný deskriptor, aktualizuje hodnotu hardvérového ukazateľa. Aktualizovaním hardvérového ukazateľa *DMA* modul oznamuje, že už použil aj odkazovanú pamäť - teda v smere *TX* už prečítal paket na odoslanie a v smere *RX* už do bloku zapísal dáta prijatého paketu. Hardvérový ukazateľ sa nachádza v pamäti na príslušnej adrese. Uloženie tohto ukazateľa v pamäti je vhodné najmä z pohľadu jeho prečítania ovládačom, ovládaču stačí prečítať hodnotu z pamäte a nemusí sa na hodnotu dotazovať opakovaným čítaním registra *DMA* modulu. Ovládač aj *DMA* modul môžu inkrementovať ukazateľe s väčším krokom, ako 1. Inkrementovanie ukazateľov s väčším krokom je žiadané, príliš časté aktualizovanie ukazateľa by zvýšilo záťaž zbernice, čo by mohlo viesť k nižšej priepustnosti. Ovládač inkrementuje softvérové ukazateľe vždy tak, aby kruhovú pamäť nepreplnil. Pri inicializácií prenosu je tiež zadaná maximálna perióda, s ktorou má *DMA* modul aktualizovať hardvérový ukazateľ.

V smere *RX* je okrem zapísania paketu potrebné uložiť aj informácie o ňom, konkrétne metadáta, jeho dĺžku a počet deskriptorov, ktoré boli použité pre jeho uloženie. Tieto informácie sú uložené v hlavičke. Veľkosť hlavičky je 32 bitov. Pre zápis hlavičiek je použitá kruhová pamäť obdobne, ako v prípade deskriptorov. V tomto prípade však hlavičky zapisuje *DMA* modul a softvérový ovládač ich číta. Aj v tomto prípade platí, že pri inicializovaní je veľkosť kruhovej pamäte daná bitovou maskou a softvérový aj hardvérový ukazateľ sú inicializované na 0. Po prijatí paketu a jeho následnom zapísaní do pamäte zapíše *DMA* modul jeho hlavičku. Následne aktualizuje hardvérový ukazateľ. Ovládač aktualizuje softvérový ukazateľ po prečítaní hlavičky.

Systém *DMA* umožňuje prenos paketov niekoľkými kanálmi. Kanály prenášajú pakety v oboch smeroch paralelne. Ovládané sú zo softvéru prostredníctvom konfiguračných registrov. Pri komunikácii s kanálmi sa používajú princípy *NPP* popísané vyššie. Každý z kanálov môže byť nezávisle aktivovaný, nakonfigurovaný na prenos konkrétnych paketov a po prenose sa môže vyžiadať jeho zastavenie. Pri spustení sa kanálu nainicializujú potrebné vyrovnávacie pamäte pre deskriptory a hlavičky. Po spustení prebieha komunikácia predávaním ukazateľov do týchto pamätí. Zastavenie sa vyžiada zápisom do príslušného registra. Systém bol navrhnutý tak, aby podporoval čo najvyšší počet *DMA* kanálov. Počet

kanálov by v ideálnom prípade nemal mať vplyv na výslednú priepustnosť. Každý kanál v smere *TX* používa sadu registrov a prípadne aj vstupnú vyrovnávaciu pamäť. Vyrovnávacia pamäť by mala byť schopná uložiť pakety o maximálnej veľkosti. Podpora vysokého počtu *DMA* kanálov potom vedie k zvyšovaniu použitých zdrojov na *FPGA*. Zvyšuje sa aj komplexnosť prvkov architektúry, ktoré pracujú so všetkými *DMA* kanálmi. Pre redukciu zdrojov používa systém koncept vybraných *DMA* kanálov. Vybrané *DMA* kanály sú taká podmnožina kanálov, ktoré sú spoločne aktívne v jednom okamihu. Maximálny počet kanálov môže byť výrazne menší ako celkový počet podporovaných kanálov. Platí, že od istého počtu kanálov sa ich zvyšovaním prestane zvyšovať priepustnosť. To znamená, že existuje horné obmedzenie počtu kanálov, ktoré musia bežať naraz, aby sa využila plná priepustnosť. Týmto spôsobom možno obmedziť komplexnosť niektorých logických operácií a znížiť množstvo použitých zdrojov *FPGA*. Podporované *DMA* kanály sa mapujú na vybrané kanály dynamicky. Pri použití vybraných *DMA* kanálov by mali byť zabezpečené nasledujúce podmienky:

- počet vybraných *DMA* kanálov je dostatočne vysoký, aby *DMA* modul dokázal posielaním požiadavkov na čítanie plne vyťažiť priepustnosť pripojenej zbernice *PCIe*,
- všetky spustené kanály sa pravidelne striedajú v roli aktívnych kanálov a nedochádza tak k vyhladovaniu,
- réžia spojená s prepínaním aktívnych kanálov nie je tak vysoká, aby došlo k poklesu vyťaženia zbernice,
- nemalo by dochádzať k tomu, že sa bude v jednom čase prepínať viacero *DMA* kanálov, čo by viedlo k dočasnému poklesu priepustnosti na zbernici *PCIe*.

Výsledný počet vybraných *DMA* kanálov je určený ako kompromis medzi dosiahnutím maximálnej priepustnosti a spotreby zdrojov *FPGA*. Vybrané kanály sú prepínané na základe nasledujúcich podmienok:

- kanál momentálne nemá žiadne dáta k odoslaniu,
- kanál bol zastavený softvérom. V tomto prípade kanál ešte dokončí prenosy paketov, ktoré sú popísané v príslušných deskriptoroch,
- pre vybraný kanál vypršal čas, po ktorý je aktivovaný. Každý kanál má pri spustení nastavený čítač *Timeout*, ktorý je implementovaný za cieľom zabránenia vyhladovenia kanálov čakajúcich na aktiváciu.

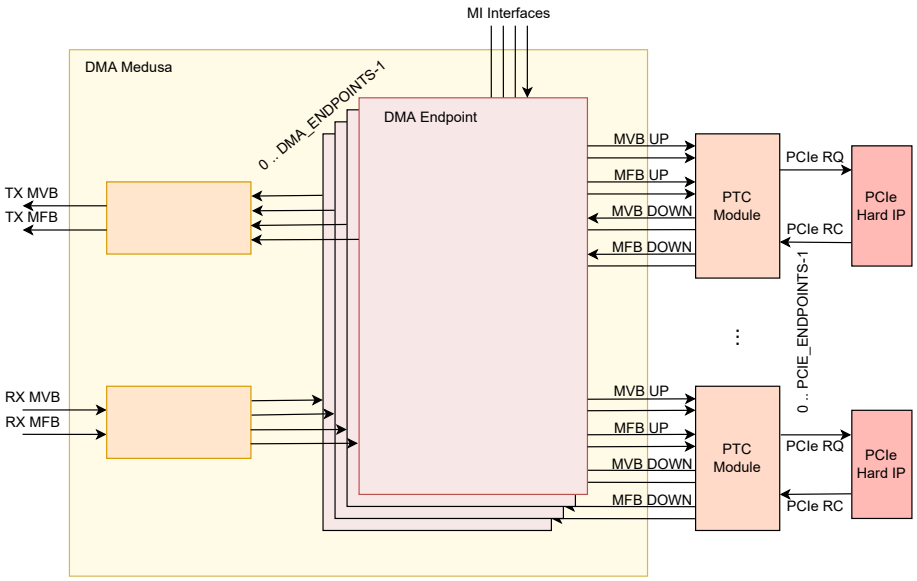
Koncept vybraných *DMA* kanálov je použitý v smere *TX*. V smere *RX* je sieťovou komponentou pre každý paket určený *DMA* kanál. *DMA* modul tak musí aktívne prijímať pakety pre všetky kanály.

4.3 Štruktúra systému

Ako už bolo spomenuté, systém *DMA Medusa* prenáša pakety medzi sieťovou komponentou a hostiteľskou pamäťou. Na strane sieťovej komponenty systém prenáša pakety pomocou komunikačných kanálov *RX* pre prijímanie a *TX* pre odosielanie. Kanál pre každý smer je implementovaný dvojicou zberníc *MFB* a *MVB*. Zbernicou *MFB* sa prenášajú dáta paketov a zbernicou *MVB* príslušné metadáta. Použitie týchto komunikačných kanálov dosahuje

veľkú priepustnosť. Aby sa táto priepustnosť zachovala aj pri komunikácii s hostiteľskou pamäťou, je použitá sada nezávislých komunikačných kanálov *PCIe*. Platforma *FPGA* implementuje tieto komunikačné kanály ako koncové body *PCIe* (*PCIe Endpoint*). Systém bol pre zachovanie priepustnosti navrhnutý tak, aby celková šírka zberníc koncových bodov *PCIe* bola rovnaká ako šírka zbernice *MFB*. Systém pracuje tak, že v smere *TX* číta pakety z niekoľkých koncových bodov *PCIe* a zapisuje ich na jedno spoločné rozhranie *MFB*. V smere *RX* prijíma všetky pakety jedným rozhraním *MFB* a pakety následne distribuuje na paralelné rozhrania koncových bodov *DMA*.

Z dôvodu dodržania požadovaného časovania je potrebné minimalizovať logiku, ktorá pracuje so všetkými koncovými bodmi *PCIe*. Systém *DMA* je za týmto účelom rozdelený na koncové body *DMA*. Spoločná časť systému obsahuje predovšetkým logiku, ktorá spája/rozdeľuje dáta medzi koncovými bodmi. Koncové body sú navrhnuté tak, že spravujú určitý počet vybraných a podporovaných *DMA* kanálov. Koncept vybraných a podporovaných *DMA* kanálov je implementovaný pre každý koncový bod. Každý kanál má priradené pole registrov v rámci koncového bodu. Pomocou registrov ovláda softvér kanály spravované koncovým bodom. Pole registrov je definované oddelene pre smery *TX* aj *RX*. Do registrového pola sa pristupuje rozhraním *MI*, ktoré je súčasťou každého koncového bodu. Pri komunikácii s kanálmi sa používajú princípy *NPP*. Koncové body *DMA* nie sú mapované na koncové body *PCIe* vzťahom $1:1$. Platí, že viacero koncových bodov *DMA* môže používať jeden koncový bod *PCIe*. Prepojenie medzi koncovými bodmi *DMA* a koncovými bodmi *PCIe* je realizované komponentou *PCIe Transaction Controller* (*PTC*). Diagram 4.1 znázorňuje komunikačné rozhrania systému *DMA Medusa*. V diagrame sú znázornené koncové body *DMA* a smer, ktorým dáta v systéme prúdia. V diagrame sú tiež zapojené moduly *PTC*, ktoré komunikujú s koncovými bodmi *PCIe*.



Obr. 4.1: Komunikačné rozhrania modulu *DMA Medusa*.

Ako už bolo spomenuté, komunikačným kanálom na užívateľskej strane v smere *RX* sa prijímajú pakety. Rozhraním *MVB* sa prenáša informácia o dĺžke paketu, metadáta, príznak zahodenia a kanál, ktorý sa má použiť pre spracovanie. Metadáta sú po prijatí paketu zapísané do vyrovnávacej pamäte hlavičiek pre daný kanál. Položky sú rozhraním prenášané štruktúrovane (použitím separátnych portov). Transakcie medzi rozhraniami *MFB* a

MVB sú parované na základne rovnakého poradia ich prijatia. Rozhraniami v smere *TX* sú odosielané pakety pre všetky *DMA* kanály. Transakcie medzi rozhraniami *MFB* a *MVB* sú rovnako párované na základe poradia. Rozhraním *MVB* sa prenášajú informácie o dĺžke odoslaného paketu, metadáta a kanál, ktorým boli odoslané. Prenášajú sa štruktúrovane.

Každý koncový bod *DMA* obsahuje rozhranie *MI*, rozhrania komunikačných kanálov pre odosielanie pamäťových požiadavkov modulu *PTC* a následné prijímanie odpovedí. Rozhranie *MI* je vytvorené pre jednoduchú manipuláciu s registrami zo softvéru. Rozhranie povoľuje reťazenie operácií. Odpovede sú vracané len u požiadavkov na čítanie. Porty rozhrania sú definované v jeho špecifikácii [10]. Adresový priestor registrov daného koncového bodu je mapovaný v hostiteľskej pamäti. Prístupy do tohto adresového priestoru sa prekladajú na požiadavky rozhrania *MI* pomocou komponenty *MTC* (*MI Transaction Controller*).

Koncový bod *DMA* zasiela pamäťové požiadavky kanálom *Up*, ktorý sa skladá z rozhraní *MFB* a *MVB*. Rozhraním *MVB* sa posielajú hlavičky požiadavkov. Rozhraním *MFB* sa posielajú zapisované dáta. U požiadavkov na čítanie sa na *MFB* negeneruje žiadna transakcia. Hlavička požiadavku obsahuje adresu, veľkosť žiadaných dát, informáciu, či ide o čítací/zápisový požiadavok, položky *FirstIB* a *LastIB* pre bajtové zarovnanie, položky *Tag* a *UnitID* (identifikátor jednotky). Identifikátor jednotky je použitý modulom *PTC* pre vytvorenie hodnoty *Tag* požiadavku *PCIe*.

Koncový bod DMA prijíma odpovede k požiadavkom na čítanie kanálom *Down*, ktorý obsahuje rozhrania *MFB* a *MVB*. Podobne ako v prípade *PCIe* transakcií platí, že odpovede na rôzne požiadavky môžu prichádzať mimo poradia a odpoveď na jeden požiadavok môže byť rozdelená na viac častí. Rozhraním *MFB* sa prijímajú rámce s prečítanými dátami. Na rozhraní *MVB* sa príjimajú položky obsahujúce informácie k týmto dátam. Prenesené informácie sú dĺžka príslušného dátového rámca na rozhraní *MFB*, informácia, či ide o posledný rámec odpovede, hodnoty *Tag* a *UnitID*. Dáta sú na rozhraní *MVB* prenášané serializovane.

Ako už bolo spomenuté, modul *PTC* (*PCIe Transaction Controller*) prekladá komunikáciu medzi koncovým bodom *DMA* a koncovým bodom *PCIe*. Rôzne platformy *FPGA* poskytujú *IP* bloky s odlišnými rozhraniami, typicky *AXI* alebo *Avalon*. Komponenta *PTC* abstrahuje koncové body *DMA* od konkrétneho rozhrania koncového bodu *PCIe*. Komponenta teda implementuje komunikáciu na transakčnej vrstve.

Kapitola 5

Predošlá verifikácia systému DMA Medusa

Systém *DMA Medusa* bol už verifikovaný. Verifikačné prostredie vykonávalo funkčnú verifikáciu. Prostredie bolo vytvorené v rámci združenia *CESNET* [6]. Implementované bolo v jazyku *SystemVerilog*, s použitím internej metodiky. Prostredie je funkčné a používané. Združenie *CESNET* však prechádza u funkčnej verifikácie k metodike *UVM*. Preto existuje snaha vytvoriť pre systém *DMA* nové prostredie podľa tejto metodiky. Nové prostredie by malo minimálne implementovať funkcionalitu starého prostredia, konkrétne generovať rovnaký stimul a používať minimálne rovnako silné kontrolné mechanizmy. Táto kapitola obsahuje analýzu pôvodného verifikačného prostredia. Verifikačné prostredie bude popísané rovnakým spôsobom, akým bola vykonaná analýza. Teda najskôr bude ukázané zapojenie verifikovaného systému a architektúra prostredia. V rámci architektúry prostredia budú ukázané aj jednotlivé komponenty, ktoré boli použité pre generovanie stimulu a komunikáciu s verifikovaným systémom. U generovania stimulu budú vymenované implementované sekvencie. Tento krok je dôležitý pre celkové pochopenie, aké mechanizmy generovania boli použité pri verifikácii. V ďalšom kroku sú popísané použité kontrolné mechanizmy, najmä funkcionalita komponenty *Scoreboard*. V poslednom kroku sú vymenované vytvorené testy, ktoré vznikli kombinovaním sekvencií a rôznych parametrov generovania.

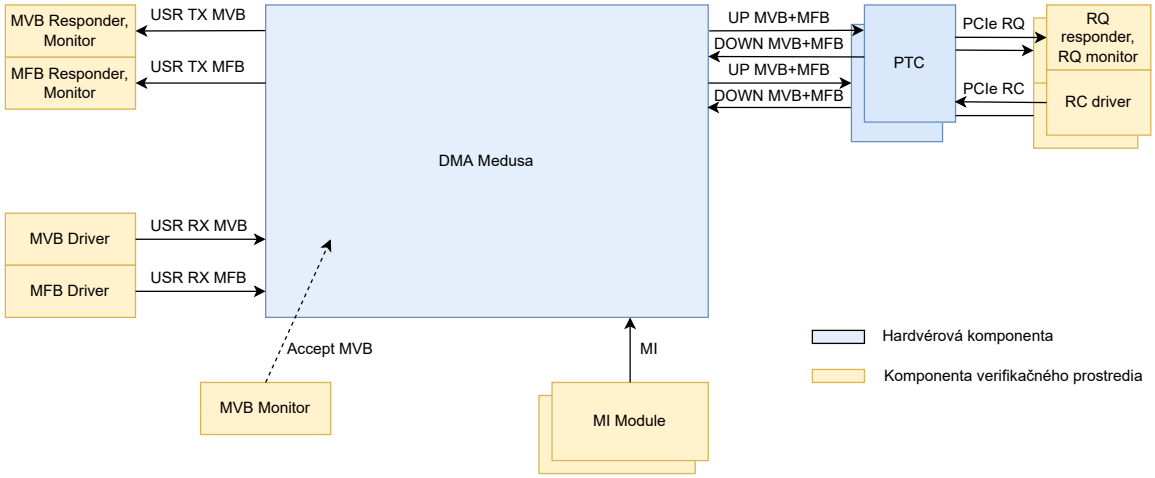
5.1 Popis verifikačného prostredia

Verifikačné prostredie je generické. Podporuje rôzny počet podporovaných aj vybraných *DMA* kanálov, rôzne šírky rozhraní v smere ku sieťovej komponente aj ku hostiteľskej pamäti. Prostredie podporuje 1 až 8 koncových bodov *PCIe*. Na jeden koncový bod *PCIe* môžú byť napojené 1 alebo 2 koncové body *DMA*. Pri verifikácii sú použité aj komponenty *PTC*.

Hlavnou úlohou verifikačného prostredia bolo generovanie stimulu a kontrola výstupov, či spĺňajú očakávané hodnoty. Na rozhrania so sieťovou komponentou pre príjem sa generujú náhodné pakety. Ďalej sa konfigurujú jednotlivé kanály prostredníctvom rozhrania *MI* tak, ako by to robil softvérový ovládač. Softvérový ovládač tiež pripravuje nové deskriptory a alokuje pamäťové bloky. Verifikačné prostredie potom prijíma požiadavky od komponent *PTC* a generuje príslušné odpovede. Generuje tiež hodinové signály a hodnotu signálu *Reset*. Pre kontrolu výstupov monitoruje výstupné rozhrania a analyzuje výstupné vektory. Hlavným kontrolným mechanizmom je komponenta *Scoreboard*, ktorá kontroluje, či boli v

smere *TX* správne odoslané všetky pakety a či boli v smere *RX* do pamäte správne zapísané prijaté pakety.

Verifikačné prostredie je napojené na systém *DMA* pomocou rozhraní, na ktoré sú napojené verifikačné komponenty. Verifikačné komponenty sú implementované ako triedy. Diagram 5.1 ilustruje zapojenie systému v prostredí. V diagrame sú zvýraznené verifikované hardvérové komponenty a komponenty prostredia. Na prepojeniach medzi komponentami sú pomenované použité rozhrania.



Obr. 5.1: Napojenie systému *DMA Medusa* do pôvodného verifikačného prostredia.

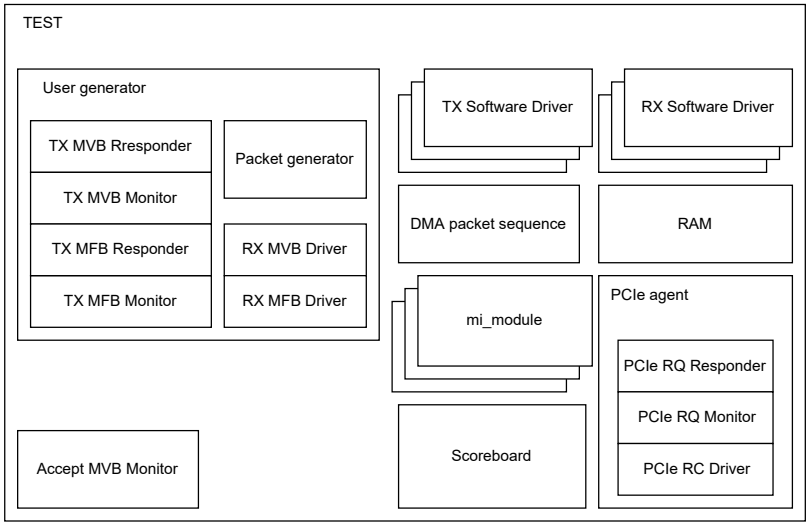
Systém *DMA* používa niekoľko zdrojov hodinového signálu. Tieto signály sú generované aj v rámci prostredia. Aktívna hodnota *Reset* sa nastavuje vždy na začiatku testu.

Na strane sieťovej karty sa nachádzajú rozhrania pre odosielanie a príjem paketov. Rozhrania pre odosielanie sú napojené na komponenty *Responder* a *Monitor*. Komponenta *Responder* generuje hodnotu pre port *Destination Ready*. Komponenta *Monitor* posiela prečítané dáta do komponenty *Scoreboard*. Rozhrania pre príjem sú napojené na komponenty *Driver*. Tie zapisujú generované pakety komponentou *User Generator*. Okrem toho je naviac do verifikačného prostredia pripojené aj mikroarchitekturálne rozhranie *MVB*. Na toto rozhranie je pripojená komponenta *Monitor*, ktorá slúži pre detekciu, či bol paket systémom skutočne prijatý, alebo bol zahodený.

Na hostiteľskej strane je systém *DMA* napojený na komponenty *PTC* a verifikačné moduly *MI*, ktoré slúžia pre konfiguráciu *DMA* kanálov. Moduly *PTC* sú napojené na verifikačné komponenty *PCIe*. Rozhranie *PCIe RQ* komunikuje s komponentami *RQ Responder* a *RQ Monitor*. Rozhranie *PCIe RC* komunikuje s komponentou *RC Driver*. Komponenta *RC Driver* zasiela odpovede na čítacie požiadavky z rozhrania *PCIe RQ*. Verifikačné moduly *MI* sú ovládané z príslušných verifikačných komponent, ktoré predstavujú softvérový ovládač.

Hlavný modul prostredia inštancuje systém *DMA* a programový blok *Test*, ktorý zabaľuje všetky verifikačné komponenty. Diagram 5.2 zobrazuje komponenty, ktoré sa nachádzajú v bloku *Test*. Komponenta *User Generator* zabezpečuje komunikáciu s rozhraniami na užívateľskej strane. O ovládanie kanálov sa starajú softvérové ovládače *RX Software Driver* a *TX Software Driver*. Softvérové ovládače používajú pre komunikáciu so systémom model pamäte *RAM* a verifikačné moduly *MI*. Sekvencia *DMA Packet* generuje pakety pre smery *TX* aj *RX*. V smere *RX* ju používajú priamo rozhrania *MVB*/*MFB* na užívateľskej

strane, v smere *TX* ju používa *TX Software Driver*. *PCIe* agent je používaný pre komunikáciu s modulmi *PTC*, odpovede na požiadavky generuje s použitím modelu pamäte *RAM*. Komponenta *Scoreboard* kontroluje správne chovanie systému.



Obr. 5.2: Základné komponenty predošlého verifikačného prostredia systému *DMA*.

Model pamäte *RAM* obsahuje metódy pre alokáciu, uvoľnenie, čítanie a zápis do pamäte. Pamäťový model používajú softvérové ovládače a *PCIe* agent. Verifikačné prostredie obsahuje jednu inštanciu modelu pamäte. Model alokuje adresy blokov náhodne tak, aby sa bloky neprekrývali.

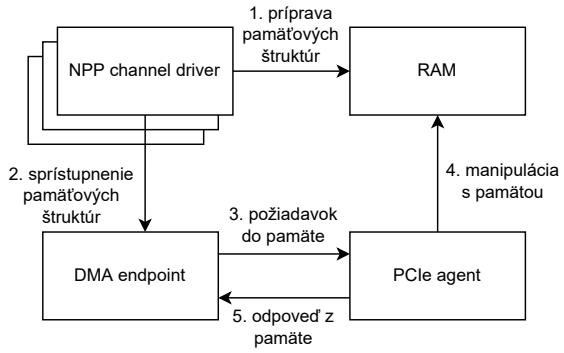
Komponenta *User Generator* zapuzdruje komponenty pracujúce s rozhraniami na strane sieťovej karty. Pre smer RX sa stará o generovanie paketov. Vygenerované pakety sú potom vložené na rozhrania *MVB* a *MFB* príslušnými komponentami *Driver*. Komponenty *Driver* vkladajú vygenerované pakety na rozhranie s náhodnými medzerami medzi položkami v rámci jedného dátového slova a náhodnými latenciami medzi slovami. U paketov sa náhodne generuje ich obsah, dĺžka, príznak zahodenia a číslo kanála, ktorý má vygenerovaný paket spracovať. Verifikačné prostredie obsahuje okrem základnej náhodnej sekvencie pre generovanie paketov aj ďalšie 3 sekvencie:

- sekvenciu *Round robin*,
- sekvenciu *Bit activation*,
- sekvenciu *All running*.

Sekvencia *Round robin* generuje v cykle jeden paket pre každý kanál. Sekvencia *Bit activation* generuje pakety len pre spustené kanály a sekvencia *All running* generuje pakety, len keď sú všetky kanály spustené.

Prostredie používa softvérový ovládač pre každý koncový bod *DMA* a pre každý jeho smer. Softvérový ovládač potom obsahuje separátne softvérové ovládače pre každý kanál. Diagram 5.3 ilustruje, ako softvérový ovládač kanála komunikuje s koncovým bodom *DMA*. Komunikácia prebieha nasledovne:

1. Ovládač kanála alokuje v pamäti bloky. V smere *TX* ich naplní paketovými dátami. Do kruhovej vyrovnávacej pamäte zapíše deskriptory alokovaných blokov.



Obr. 5.3: Komunikácia softvérového ovládača kanála s koncovým bodom *DMA*.

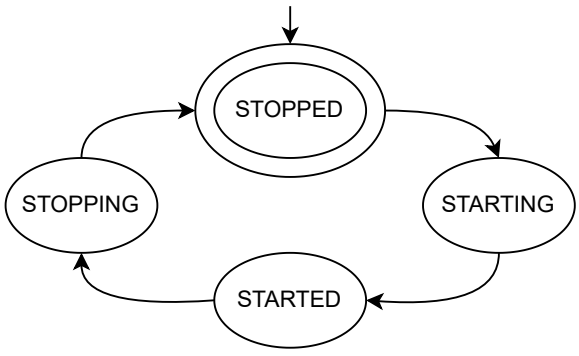
2. Ovládač kanála sprístupní deskriptory a alokované bloky koncovému bodu zvýšením softvérového ukazateľa do vyrovnávacej pamäte deskriptorov.

3. Koncový bod následne prečíta nové deskriptory a podľa nich zapíše prijaté pakety do blokov v smere *RX*, v smere *TX* prečíta pakety a pošle ich na rozhrania na užívateľskej strane. Čítanie a zápis prebieha pomocou *PCIe* agenta, ktorému koncový bod najskôr pošle požiadavku.

4. *PCIe* agent prijme požiadavku do pamäte a prečíta/zapíše dáta do pamäťového modelu *RAM*.

5. *PCIe* agent v prípade požiadavku na čítanie vráti prečítané dáta.

Ovládače pre jednotlivé kanály môžu bežať sériovo alebo paralelne. Ovládač jedného kanálu je implementovaný pomocou stavového automatu. Diagram 5.4 zobrazuje stavy softvérového ovládača. V počiatočnom stave je kanál zastavený. V stave *Starting* prebieha inicializácia kanála. V stave *Started* ovládač zotrvá, kým je kanál spustený. Kanál ostane spustený náhodný čas. Po prechode do stavu *Stopping* ovládač dokončí začaté prenosy a začne príslušný kanál zastavovať. Po zastavení kanála prejde ovládač do stavu *Stopped* a zotrvá v ňom náhodnú dobu.



Obr. 5.4: Stavy softvérového ovládača jedného kanála.

Softvérový ovládač kanála v smere *RX* pracuje v stavoch ilustrovaných v diagrame 5.4 nasledovne. Po prejdení do stavu *Starting* najskôr vygeneruje a inicializuje potrebné vyrovnávacie pamäte pre deskriptory, hlavičky a hardvérové ukazatele. Veľkosť vyrovnávacej

pamäte pre deskriptory a hlavičky je daná konfiguráciou testu. Následne ovládač alokuje počiatočný počet pamäťových deskriptorov. Následne spustí kanál inicializáciou jeho registrov prostredníctvom rozhrania *MI*. Po spustení kanála prejde do stavu *Started*. Ovládač v tomto stave cyklicky náhodne alokuje ďalšie pamäťové bloky, pripravuje deskriptory do pamäte, číta prijaté pakety a ich hlavičky. Typy deskriptorov sú generované náhodne, s ohľadom na popisovanú pamäť. Po prečítaní hlavičiek a paketových dát aktualizuje softvérové ukazatele do vyrovnávacích pamätí. Následne náhodne uvoľňuje použitú pamäť. Prečítané pakety sú zaslané do komponenty *Scoreboard*. Po prejdení do stavu *Stopping* ovládač najskôr vyžiada zastavenie kanála zápisom do registra *Control*. Následne pokračuje v čítaní prijatých dát, až kým sa kanál nezastaví. Nakoniec uvoľní alokovanú pamäť.

Softvérový ovládač kanála v smere *TX* v stave *Starting* pracuje podobne v smere *RX*. Najskôr alokuje priestor pre vyrovnávaciu pamäť deskriptorov a hardvérový ukazateľ. Následne alokuje niekoľko pamäťových blokov pre odosielanie paketov. Potom inicializuje registre daného kanála a kanál spustí. V stave *Started* ovládač cyklicky alokuje dodatočnú pamäť, ak to je potrebné. Následne získa paket z paketovej sekvencie. Pakety generuje sekvencia *TX packet sequence*. Ďalej začne ovládač generovať pamäťové deskriptory a paket začne zapisovať do pamäte. Zapísané pakety sú zaslané komponente *Scoreboard*. Do deskriptorov je zapísaná tiež dĺžka a metadáta paketu. Typy deskriptorov sú generované náhodne. Po zapísaní paketu ovládač prečíta aktuálny hardvérový ukazateľ a náhodne uvoľní použitú pamäť. Následne ovládač sprístupní vygenerované deskriptory *DMA* kanálu. Po prejdení do stavu *Stopping* prestane generovať nové deskriptory s paketmi na odoslanie a dokončí sprístupnenie všetkých deskriptorov, ktoré sú už nachystané. Po ich sprístupnení a prijatí *DMA* modulom vyžiada zastavenie kanála zápisom do registru *Control*. Potom počká, kým sa kanál zastaví. Po zastavení uvoľní použitú pamäť.

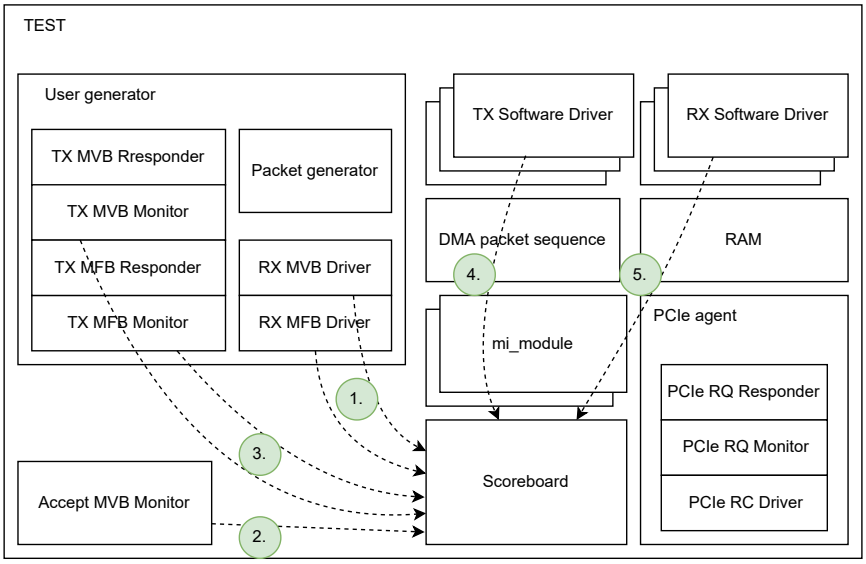
Komponenta PCIe agent odbavuje požiadavky na prístup do pamäte. Do pamäte pristúpi okamžite, odpovede požiadavkov na čítanie náhodne rozdelí na viac častí a jednotlivé časti odpovedí premieša. Časti odpovedí sú vrátené s náhodným opozdením.

5.2 Kontrolné mechanizmy

Hlavným kontrolným mechanizmom verifikačného prostredia je komponenta *Scoreboard*. Táto komponenta v priebehu testu kontroluje či *DMA* modul prečítal/zapísal pakety správne. Na konci testu skontroluje, či je počet prenesených paketov s očakávaným počtom. Verifikačné prostredie obsahuje aj ďalšie kontrolné mechanizmy v jednotlivých komponentách. Model pamäte kontroluje prístupy koncových bodov mimo alokovanej pamäte. Na konci testu sa naviac kontroluje, či sa všetky kanály zastavia do vypršania času *Timeout*. Nezastavenie môže indikovať ich zaseknutie.

Komponenta *Scoreboard* komunikuje s ostatnými komponentami prostredia prostredníctvom volaní *Callback*. Tieto volania vykonávajú komponenty *Monitor* na rozhraniach k sieťovej komponente a softvérové ovládače jednotlivých kanálov. V diagrame 5.5 sú ilustrované použité volania. Prenášajú nasledujúce dáta:

1. pakety zapísané na *RX* rozhrania *MVB* a *MFB* na užívateľskej strane,
2. pakety prijaté systémom *DMA Medusa*, ktoré neboli zahodené,
3. odoslané pakety systémom *DMA Medusa* rozhraniami *MVB* a *MFB* na užívateľskej strane,
4. pakety zapísané do pamäte softvérovým ovládačom – pakety sú určené na odoslanie,



Obr. 5.5: Komunikácia komponenty *Scoreboard* s ostatnými komponentami prostredia.

5. prijaté pakety prečítané softvérovým ovládačom z pamäte.

Komponenta *Scoreboard* inštancuje pre každý koncový bod *DMA* triedy, ktoré kontrolujú prenesené pakety. Triedy sú definované pre každý smer. Keď softvérový ovládač daného kanála v smere *TX* zapíše paket do pamäte, pošle ho do komponenty *Scoreboard*, kde sa uloží do frontu *FIFO*. Na užívateľskej strane sú potom pakety prečítané na užívateľských rozhraniach *TX MVB* a *MFB* komponentami Monitor. Transakcie sa medzi rozhraniami *MVB* a *MFB* párujú pomocou frontu *FIFO*. Po spárovaní transakcií sa paket aj s jeho metadátami porovná s prvým paketom vo fronte softvérového ovládača pre daný kanál.

V smere *RX* prijíma modul *DMA* pakety bez ohľadu na to, či sú príslušné *DMA* kanály spustené. Ak je paket určený pre kanál, ktorý je aktuálne vypnutý, paket bude zahodený. Po spustení kanála modul *DMA* prestane zahadzovať pakety pre tento kanál a začne ich zapisovať do pamäte. Pakety prijaté systémom *DMA* môžu byť zahodené v nasledujúcich prípadoch:

- došlo k preplneniu niektorej vnútornej vyrovnávacej pamäte a bol povolený neblokujúci mód prijímania (generický parameter *Blocking Mode* bol nastavený do log.0),
- paket bol sieťovou komponentou určený na zahodenie,
- *DMA* kanál bol vypnutý,
- *DMA* kanál sa začal spúšťať, ešte však nezačal prijímať pakety,
- *DMA* kanál sa začal vypínať a tak začal zahadzovať pakety.

Najmä v posledných dvoch prípadoch nie je z pohľadu vonkajších rozhraní jasné, kedy začne/prestane dochádzať k zahadzovaniu, preto je použité mikroarchitekturálne rozhranie *MVB*. Na toto rozhranie sa zapisuje informácia o každom pakete v smere *RX*, či sa paket zahodí, alebo bude uložený do pamäte.

V smere *RX* je najskôr paket prijatý na užívateľskej strane rozhraniami *MVB* a *MFB* v smere *RX*. Transakcie medzi týmito rozhraniami sa párujú pomocou frontov *FIFO*. Ak bol

pre prijatý paket nastavený bit *Discard* do log. 1, paket má byť zahodený. Po prijatí paketu sa následne získa informácia z rozhrania *Accept MVB*, či je paket určený na zahodenie. Ak paket nebol označený na zahodenie a pri tom mal na rozhraní *MVB RX* nastavený bit *Discard* do log.1, verifikačné prostredie sa zastaví na chybe. Inak v prípade, že paket nebol označený na zahodenie, uloží sa do frontu *FIFO* daného kanála. Keď systém *DMA* zapíše paket do pamäte a softvérový ovládač daného kanála ho vyčíta, pošle sa do komponenty *Scoreboard*, kde sa porovná s prvým paketom vo fronte pre príslušný kanál.

5.3 Implementované testy

Verifikačné prostredie obsahuje rôzne pripravené testy. Tieto testy sú vytvorené kombinovaním rôznych sekvencií, rôznych parametrov generovania stimulu a takisto rôznym nastavením generických parametrov hardvérového modulu *DMA*. Zmena parametrov hardvéru umožňuje lepšie otestovanie konkrétnych častí. Základný test obsahuje predvolené hodnoty parametrov nastavené tak, aby bol generovaný stimul čo najviac generický. Ďalšie testy boli vytvorené zmenou týchto parametrov. V testoch sú upravované nasledujúce parametre:

- Doba spusteného a zastaveného softvérového ovládača kanála. Presná doba zastaveného a spusteného kanála je vygenerovaná pre každú iteráciu. Základný test necháva kanál spustený po 800 – 8 000 iterácií, zastavený po 100 – 500 iterácií.
- Veľkosť paketov. Niektoré testy obmedzujú veľkosť generovaných paketov. Základný test generuje pakety o veľkosti 64 – 8 192 bajtov.
- Počet prenesených paketov každým kanálom. Základný test prenáša 2 000 paketov v každom smere.
- Veľkosť medzier medzi dátovými rámcami na rozhraniach. Medzery medzi dátovými rámcami sa generujú náhodne. Naviac, verifikačné prostredie dynamicky prepína medzi režimami, v ktorých sa veľkosti medzier generujú. Prostredie podporuje režimy pre generovanie malých, stredných, veľkých a naviac generovanie len maximálnych alebo žiadnych medzier. Ďalšie testy potom používajú len niektoré režimy. Režimy sa používajú tiež pre generovanie hodnoty portu *Destination Ready* pri potvrdzovaní prijatia dátových slov.
- Veľkosť vyrovnávacích pamätí pre uloženie deskriptorov a hlavičiek. Veľkosť vyrovnávacích pamätí je pevne definovaná na začiatku testu. Základný test používa 128 položiek pre obe pamäte.
- Taktovacia frekvencia hodín pre bloky pracujúce s rozhraniami sieťovej karty a rozhraniami smerom k zbernici *PCIe*. Rýchlosť taktovania sa udáva periódou. Základný test používa periódu 5 ns pre obe rozhrania.
- Nastavenie generického parametru *Blocking Mode* pre povolenie/zakázanie zahadzovania paketov smere *RX*, ak dôjde k preplneniu interných vyrovnávacích pamätí. Základný test zahadzovanie zakazuje.
- Počet koncových bodov *DMA* na jeden koncový bod *PCIe*. Základný test generuje 1 koncový bod *DMA* na 1 koncový bod *PCIe*.
- Počet koncových bodov *PCIe*. Základný test používa 1 koncový bod *PCIe*.

- Počet *MFB* regiónov na rozhraniach k sieťovej karte. Základný test generuje 1 región pre každý koncový bod *DMA*.
- Generický parameter *TX Usr MFB Gap Size*. Tento parameter určuje, s akou priemernou medzerou má hardvérová komponenta *Packet Planner* zapisovať pakety na rozhranie *MFB* pri odosielaní paketov do sieťovej komponenty. Základný test nastavuje medzeru na 26 blokov.

Testy možno zoskupiť podľa oblasti, na ktorú sa zameriavajú. V každej skupine sú potom definované rôzne ďalšie parametre, aby sa daná oblasť otestovala čo najlepšie.

Základné testy sú určené pre otestovanie základnej logiky pri rôznych generických parametroch modulu *DMA*. Testy manipulujú s taktovacou frekvenciou, počtom koncových bodov a počtom regiónov. Konkrétne nastavujú periódu hodín pre rozhrania so sieťovou kartou na 7 ns, periódu pre rozhrania so zbernicou *PCIe* na 3 ns, 2 koncové body *PCIe*, 2 koncové body *DMA* na každý koncový bod *PCIe* a 1 región u rozhraní *MFB* na strane sieťovej karty.

Pokročilé testy pracujú s parametrom *Blocking Mode* a manipulujú s počtom *TX* aj *RX* kanálov. Testy ďalej nastavujú rôzne doby spustených a zastavených kanálov, rôzne veľké medzery medzi položkami na rozhraniach, menšie veľkosti vyrovnávacích pamätí pre deskriptory a hlavičky. Nastavuje sa aj mód *Loopback*, ktorý medzi sebou prepojí komunikačné kanály *RX* a *TX* na strane sieťovej karty. Konkrétne, test nastavuje 4 vybrané kanály a 64 podporovaných kanálov v smere *TX*. Ďalší test nastavuje 4 podporované a 4 vybrané pre jeden koncový bod v smere *TX* a 64 pre jeden bod v smere *RX*. Veľkosti vyrovnávacích pamätí sú nastavené na 32 položiek. V testoch sa nechávajú kanály spustené dlhšie (1000 – 4000 iterácií), prípadne sa testuje rýchle prepínanie medzi spustenými (10 – 100 iterácií) a zastavenými (10-50 iterácií). Rýchle prepínanie kanálov sa testuje aj v kombinácií s parametrom *Blocking Mode*. V skupine sú testy, ktoré generujú iba malé (64 – 512 B) a iba veľké pakety (1024 – 8192 B). Jeden test zakazuje generovanie medzier medzi dátovými rámcami na rozhraní *MFB* pre odosielanie paketov do sieťovej komponenty.

Testy generujúce malé pakety generujú pakety o veľkosti 64 – 512 B. Testy manipulujú s dobou spustenia kanálov. Testované sú dlho spustené kanály, pričom sa manipuluje s parametrom *Blocking Mode*. U kanálov sa tiež testuje časté prepínanie. Testujú sa tiež rôzne veľkosti medzier medzi rámcami a hodnota *Ready* na portoch potvrdzujúcich prijatie transakcií. Testujú sa tiež malé medzery medzi rámcami na rozhraniach ku zbernici *PCIe*. Niektoré testy pracujú s rôznou periódou hodín, podobne ako v prípade základných testov. Systém *DMA* sa testuje s pripojením 2 – 4 koncových bodov *PCIe*, s 2 koncovými bodmi *DMA* pre jeden koncový bod *PCIe*. Testuje sa tiež konfigurácia rozhraní *MFB* s počtom rámcov rovným dvojnásobku počtu koncových bodov *DMA*. Testuje sa tiež konfigurácia rozhrania *MFB* s rovnakou šírkou dátového slova ako v predošlom prípade, teraz však len s jedným regiónom.

Testy generujúce veľké pakety generujú pakety o veľkosti 1024 – 8192 B. Tieto testy nechávajú dlhšie spustené kanály, generujú malé medzery medzi dátovými rámcami na vonkajších rozhraniach. Testy používajú 2 koncové body *PCIe*, pričom pre každý koncový bod *PCIe* sú vygenerované 2 koncové body *DMA*. Testy tiež manipulujú s frekvenciou hodín rovnako ako v základných testoch.

Testy zamerané na rozhrania PCIe testujú konfigurácie so štyrmi koncovými bodmi *PCIe*, pričom každý koncový bod *PCIe* je napojený na dva koncové body *DMA*. Tieto testy často prepínajú kanály na zapnuté a vypnuté, pričom sa generujú minimálne medzery medzi rámcami na rozhraniach na strane sieťovej karty aj na strane zbernice *PCIe*.

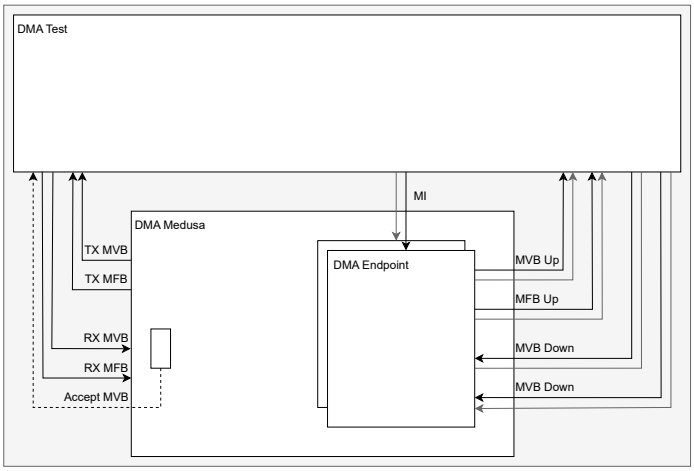
Kapitola 6

Požiadavky na vytvorené verifikačné prostredie

Úlohou práce je návrh a implementácia verifikačného prostredia systému *DMA Medusa* podľa metodiky *UVM*. Ako sa spomína v úvode, systém už bol pred tým verifkovaný, ktoré je analyzované v kapitole 5. Predošlé prostredie vhodne implementuje základné princípy paketových prenosov. Združenie *CESNET* dnes prechádza k používaniu metodiky *UVM*. Preto je ich motivácia prepísať pôvodné verifikačné prostredie. Nové prostredie by malo zachovať základnú funkcionalitu pôvodného – kontrolné mechanizmy a generované scenáre. Naviac by malo implementovať ďalšie vylepšenia, ktoré budú viesť k vyššej kvalite verifikovaného systému. Hlavným nedostatkom predošlého prostredia je absencia dokumentu, ktorý by odrážal obsah verifikačného plánu – vymenoval by oblasti, na ktoré sa verifikácia zameria a stratégiu pokrytia týchto oblastí. Verifikačný plán je typicky kolekciou plánov testov. U každého testu býva uvedená testovaná vlastnosť, akým stimulom sa má otestovať a mechanizmus, ktorým sa skontroluje. Plán by mal byť vytvorený a skonzultovaný ideálne pred vytvorením prostredia. Môže sa upravovať aj v priebehu implementácie, aby bolo možné aplikovať novozískané znalosti o systéme, reagovať na jeho zmeny, prípadne navrhnúť vylepšenie prostredia na základe odhalenej chyby. Vytvorenie dokumentu pomôže nielen pri spätnej analýze prostredia, minimalizuje aj riziko, že sa niektoré navrhnuté testy zabudnú implementovať. Naviac umožňuje lepšie udržať kvalitu pri aplikovaní zmien v systéme. Ak verifikačný inžinier aktualizuje prostredie po zmene špecifikácie systému, vďaka dokumentu môže mať lepší prehľad, či systém ostane zverifikovaný v dostatočnej miere aj po aplikovaní zmien. V neposlednom rade dokument umožní jednoduchšiu kontrolu vykonanej verifikácie treťou osobou. Pre nové prostredie sú informácie odpovedajúce obsahu verifikačného plánu vymenované v tejto kapitole. Kapitola sa zameria na definovanie rozsahu verifikácie (aké moduly sa majú verifikovať) a vlastnosti, ktoré je potrebné otestovať a kontrolovať. Vlastnosti vychádzajú primárne zo špecifikácie systému, inšpirované sú aj predošlým prostredím. Vlastnosti sú rozdelené na požadované vlastnosti generovaného stimulu a kontrolované vlastnosti. Vlastnosti generovaného stimulu sa majú implementovať ako sekvencie. Kontrolované vlastnosti sú vlastnosti systému, ktoré by mali byť dodržané v priebehu testovania. Implementovať sa majú prostredníctvom kontrolných mechanizmov.

6.1 Verifikované Hardvérové moduly

Jedným z prvých rozhodnutí pri vytvorení verifikačného prostredia je definovanie rozsahu verifikácie a množiny hardvérových modulov, ktoré budú verifikované. Nové prostredie bude komunikovať so systémom na rozhraniach sieťovej karty a rozhraniami *MI* rovnako ako pôvodné. Rozdiel bude pri zapojení hostiteľskej pamäte. Komunikácia s pamäťou prebieha prostredníctvom zbernice *PCIe*. Koncové body systému *DMA* komunikujú s koncovým bodom *PCIe* prostredníctvom modulov *PTC*. Moduly *PTC* nie sú súčasťou hlavného *DMA* modulu. Z obrázku 5.1 možno vidieť, že predošlé verifikačné prostredie používalo moduly *PTC*. Moduly sú verifikované v samostatnom prostredí. Stojí za zváženie, či by mali byť súčasťou nového verifikačného prostredia systému *DMA*. Funkcionalita modulu *PTC* je uvedená v podkapitole 4.3. Predpokladá sa, že modul *DMA* bude vždy používať moduly *PTC*. Z tohto dôvodu môže byť výhodnejšie verifikovať systém spolu s modulmi. Tým sa otestuje, že komunikácia medzi koncovými bodmi *DMA* a modulmi *PTC* funguje správne. Na druhej strane platí, že malá zmena v implementácií modulu *PTC* môže zmeniť časovanie smerom ku koncovému bodu *DMA*. V takom prípade bude potrebné znovu vykonať kompletnú verifikáciu systému *DMA* vrátane analýzy pokrytia (pokrytia kódu aj funkčného pokrytia). Systém *DMA* je veľmi komplexný, takže vykonanie verifikácie môže mať vysokú časovú náročnosť. Opačným prístupom by bolo verifikovať modul *DMA* bez použitia *PTC*. Potom by sa museli pri verifikácii pokryť všetky stavy, ktoré by mohli nastať pri komunikácii. Pre kontrolu, že naozaj boli otestované, možno implementovať funkčné pokrytie. Mali by sa pokryť nielen časové závislosti medzi jednotlivými portami rozhraní, ale aj rozličný čas vrátenia odpovedí na požiadavky, náhodné poradie, poradie ich vrátenia a rozloženie odpovede na viac častí. Potom by bolo možné predpokladať, že modul *DMA* bude nezávislý na zmenách časovania *PTC*. Aby bolo možné nezávislosť časovania potvrdiť, bude potrebné verifikovať modul *DMA* s modulmi *PTC* na vyššej úrovni. Toto prostredie by naviac mohlo používať sieťovú kartu, moduly *MTC*, prípadne aj softvérové ovládače. Modul *MTC* slúži pre komunikáciu medzi softvérovým ovládačom a rozhraním *MI*. Prostredie vytvorené v tejto práci bude verifikovať systém bez modulov *PTC*. Obrázok 6.1 zobrazuje očakávané zapojenie systému v prostredí. Prostredie by malo podporovať všetky konfigurácie systému, ako v prípade predošlého prostredia.



Obr. 6.1: Očakávané zapojenie systému *DMA* do nového verifikačného prostredia.

6.2 Požadované vlastnosti generovaného stimulu

V podkapitole 4.3 je uvedená základná štruktúra systému a spôsob jeho komunikácie okolitými komponentami. Táto podkapitola vymenuje požadované vlastnosti stimulu pre všetky jeho rozhrania. Zameria sa postupne na komunikáciu so softvérovým ovládačom, s hostiteľskou pamäťou a rozhraniami sieťovej karty. U každej vlastnosti je pridaný jej podrobný popis, prípadne vysvetlená situácia, ktorá by mala pri testovaní nastať. Ďalej je načrtnutý spôsob, akým by sa vlastnosť/situácia mala dostatočne pokryť.

Komunikácia so softvérovým ovládačom

Softvérové ovládače by mali ovládať každý z kanálov nezávisle. Kanály jedného koncového bodu *DMA* zdieľajú rozhranie *MI*. Kanály by mali byť spúšťané tak, aby sa požiadavky rôznych kanálov pre rozhranie *MI* ľubovolne kombinovali. Každý kanál by mal po spustení zapísať/prijať náhodný počet paketov a následne sa zastaviť. V krajnom prípade môže preniesť 0 paketov. Maximálny počet by mal byť daný dĺžkou testu. Vhodné je zvýšiť pravdepodobnosť situácie, kedy sa kanál niekoľko krát za sebou spustí a zastaví, pričom prenesie v každej iterácii minimálny počet paketov. Toto možno dosiahnuť generovaním počtu paketov v rámci daného prenosu vo viacerých režimoch:

- v režime, kedy kanál neprenesie žiadne pakety,
- v režime, kedy prenesie len minimálny počet paketov,
- v náhodnom režime,
- v režime, kedy prenáša vyšší počet paketov, prípadne všetky možné pakety.

Po zastavení kanála by mal ostať kanál vypnutý po náhodnú dobu. Čas by mal byť od 0 až po niekoľko sto taktov. Vhodné je však zvýšiť pravdepodobnosť krátkeho zastavenia, ktoré by mohlo mať najvyššiu šancu odhalenia prípadnej chyby vo verifikovanom systéme.

Softvérový ovládač prenáša prostredníctvom deskriptorov informáciu o pamäťových blokoch, ktoré sa majú využiť. Každý deskriptor má veľkosť 64 bitov. Deskriptory sa ukladajú do kruhovej vyrovnávacej pamäte, ktorá je alokovaná ovládačom pred začatím prenosu. Vyrovnávacia pamäť v softvéri typicky zaberá niekoľko stránok. Jedným z cieľov verifikácie je otestovať správnu aktualizáciu hardvérových ukazateľov, prípadne správnu detekciu plnosti/prázdnosti pamäte. Preto by jej veľkosť mala byť generovaná náhodne, medzi 32 a 1024 položkami (256 až 8 192 B).

V smere *TX* ovládač po spustení kanála cyklicky pripravuje deskriptory s paketmi, ktoré sa majú preniesť. Následne tieto deskriptory zapíše, aktualizuje softvérový pointer a uvoľní už použité deskriptory. Počet použitých deskriptorov určí prečítaním aktuálnej hodnoty hardvérového ukazateľa. V každom kroku by mal tak ovládač pripraviť náhodný počet paketov. Pakety môžu mať rozličnú dĺžku. Vhodné je pri generovaní dĺžky paketov aj ich počtu generovať za sebou rôzne dĺžky, napríklad množstvo malých, prípadne množstvo veľkých paketov za sebou. Pakety by mali mať dĺžku 60 - 8 192 B. Počet pripravených deskriptorov by mal byť pri každej aktualizácií softvérového ukazateľa 0 až taký počet, aby sa plne zaplnila vyrovnávacia pamäť.

Podľa podkapitoly 4.1 deskriptor v smere *TX* obsahuje informáciu o dĺžke paketu (prípadne pamäťového bloku), adresu začiatku pamäťového bloku, typ hlavičky a metadáta. V prípade, že sa paket nezmestí do aktuálneho pamäťového bloku, môže softvér zapísať paket

do niekoľkých blokov. V tomto prípade nastaví do deskriptoru príznak *Next*. Softvérový ovládač môže teoreticky zapísať niekoľko paketov do jedného pamäťového bloku. V takom prípade pre každý paket vytvorí vlastný deskriptor. Ovládač by mal preto generovať bloky o náhodnej veľkosti. Do blokov by mal zapisovať pakety, ktoré sa buď zmestia celé, prípadne len ich časť. Mal by sa tiež náhodne rozhodnúť, či do pamäťového bloku zapíše aj ďalší paket, alebo alokuje nový pamäťový blok. Pravdepodobnosť alokovania nového bloku môže byť približne 50%. Veľkosť alokovaného bloku by mala byť 60 – 8 192 B.

V smere *RX* ovládač alokuje nové pamäťové bloky, kam môže kanál zapisovať pakety. Tak ako v prípadne pripravovania dekriptorov smere *TX*, softvér môže alokovať niekoľko pamäťových blokov v každom kroku. Príslušné deskriptory sa zapíšu do vyrovnávacej pamäte. Do pamäte by sa v každom kroku malo prepísať 0 až maximálny počet deskriptorov tak, aby sa pamäť zaplnila. Veľkosť každého bloku môže byť 60 – 8 192 B. Prijatý paket sa môže, alebo nemusí zmestiť do pamäťového bloku popísaného aktuálny deskriptorom. V prípade, že sa nezmestí, môže sa so zápisom pokračovať do nasledujúceho bloku. Aby táto situácia nastávala v dostatočnej miere, zapisované pakety by mali mať vhodnú veľkosť. Po prijatí paketu modul *DMA* zapíše hlavičky prijatých paketov do vyrovnávacej pamäte hlavičiek. Softvér signalizuje prečítanie hlavičky aktualizáciou softvérové ukazateľa.

Komunikácia s pamäťou

Koncový bod *DMA* zasiela pamäťové požiadavky modulu *PTC* rozhraniami *MVB* a *MFB Up*, modul *PTC* zasiela odpovede rozhraniami *MVB* a *MFB Down*. Modul konvertuje požiadavky a príslušné odpovede medzi koncovým bodom *DMA* a koncovým bodom *PCIe*. Platí teda, že požiadavky sa môžu odbavovať v rôznom poradí a odpovede na čítanie sa vracajú rozložené na niekoľko častí podľa pravidiel zbernice *PCIe*. Odpovede sa vytvárajú len pre požiadavky na zápis. Ako sa spomína v podkapitole 4.3, hlavička požiadavku sa skladá z adresy požiadavku, veľkosti požiadavku v počte 32-bitových slov (*Dwords*), bajtového zarovnania, položky *Tag*, identifikátoru jednotky a príznaku, či sa môže využiť uvolnené poradie (*Relaxed ordering*) tak, ako ho definuje špecifikácia zbernica *PCIe*. Odpovede na čítacie požiadavky sa môžu rozdeliť na viac častí. Deliť sa musia tak, aby boli zarovnané na hodnotu parametra *RCB*. Maximálna veľkosť jednej časti je obmedzená parametrom *MPS*. Verifikačné prostredie by malo deliť každý požiadavok na náhodný počet častí. Latencia pri prístupe do pamäte je typicky niekoľko tisíc hodinových taktov. Za účelom optimalizácie času verifikácie by malo verifikačné prostredie generovať odpovede s latenciou v nižšom počte taktov, no niekoľkokrát by vygenerovaná latencia mala byť aj aspoň sto taktov. Pamäť by mala byť schopná prijať aspoň niekoľko desiatok požiadavkov naraz, kým zapíše príslušné odpovede. Zbernica *PCIe* povoľuje nasledovné predbiehanie medzi transakciami:

- zápisové transakcie sa môžu navzájom len predbiehať v prípade, že je príznak uvoľneného poradia v log.1,
- zápisová transakcia môže predbehnúť transakcie na čítanie,
- transakcie na čítanie nesmú predbiehať zápisové transakcie,
- transakcie na čítanie sa môžu navzájom ľubovoľne predbiehať.

Odpovede na čítacie požiadavky sa môžu vrátiť v nasledujúcom poradí:

- odpovede na rôzne požiadavky (s odlišnou položkou *Tag* a identifikátorom jednotky) sa môžu ľubovolne predbiehať,

- odpovede na jeden požiadavok sa vracajú v poradí. Posledný požiadavok je označený príznakom *Completed*.

Vykonanie viacerých požiadavkov na čítanie je idempotentné a tak poradie odbavenia samotných požiadavkov na čítanie by nebolo potrebné určovať náhodne. Keďže však zápisové požiadavky môžu predbehnúť transakcie na čítanie, je potrebné premiešať poradie zápisových požiadavkov s čítacími a naviac aj čítacie požiadavky medzi sebou. V prípade, že sa v rámci systému *DMA* používa príznak uvoľneného poradia v log.1, je potrebné generovať náhodné poradie aj u zápisov. Naviac platí, že špecifikácia zbernice PCIe explicitne nedefinuje granularitu transakcií na čítanie. Majme nasledujúcu situáciu:

- koncový bod DMA požiada o prečítanie z 2 lokalít jedným požiadavkom na čítanie. Požiadavok sa odbaví po niekoľkých častiach zarovnaných podľa RCB,
- koncový bod DMA požiada o zápis do lokality, z ktorej pred tým požiadal o čítanie.

V tomto prípade sa môže stať, že najskôr sa odbaví čítanie z jednej lokality, následne sa vykoná zápis a nakoniec sa vykoná čítanie aj z druhej lokality, pričom sa vrátia už aktualizované dáta. Z tohto dôvodu je potrebné, aby vytvorený model pamäte určoval nielen náhodné poradie medzi transakciami, ale aby umožnil premiešať odbavenie jednotlivých častí čítacej transakcie so zápisovými transakciami.

Veľkosť požiadavku sa určuje v počte 32-bitových slov a adresa požiadavku by mala byť zarovnaná na 4 bajty. Aby bolo možné odoslať požiadavok na nezarovnanú adresu, systém *DMA* nastavuje hodnoty parametrov zarovnania - *FirstIB*, a *LastIB*. Pri zápise nimi špecifikuje počet nevalidných bajtov na začiatku a na konci dát na zapísanie. Prečítané dáta sa rovnako vracajú zarovnané na 4 bajty, nevalidné bajty by sa mali vygenerovať náhodne.

Na úrovni rozhraní *MFB* a *MVB* je potrebné pokryť rôzne situácie pri prenose. Pri prijímaní požiadavkov je potrebné náhodne nastavovať hodnoty pre porty *Destination ready*. Hodnoty by sa mali generovať vo viacerých režimoch, u ktorých sa hodnota prepína rôzne často. Pri odpovedi môžu byť hlavičky a príslušné dáta rozlične oneskorené tak, že najskôr dorazí hlavička, následne dáta a opačne, najskôr hlavička, potom dáta. Táto vlastnosť by sa mala otestovať dostatočne náhodnými latenciami medzi zapisovanými slovami na rozhrania. Funkčné pokrytie by malo merať počet hlavičiek zapísaných bez príslušných dát a naopak, počet zapísaných dátových rámcov bez príslušných hlavičiek. Na rozhraní *MVB Up* by sa malo naviac merať, či sa pokryli všetky kombinácie parametrov zarovnania a veľkosti požiadavku. U požiadavku je minimálna veľkosť po zohľadnení zarovnania teoreticky 1 bajt, maximálna je daná parametrami *MRRS* (požiadavok na čítanie) a *MPS* (požiadavok na zápis).

Rozhrania na strane sieťovej karty

Rozhraniami k sieťovej karte prúdia pakety v smere *RX* a *TX*. Pre každý smer sa na prenos paketových dát používa dvojica rozhraní *MFB* a *MVB*. V smere *RX* sú generované pakety priamo zapisované verifikačným prostredím. V smere *TX* sú zapísané *DMA* modulom pakety, ktoré boli pred tým vygenerované a zapísané do pamäte sieťovými ovládačmi. Požiadavky na generovaný stimul tu možno definovať na úrovni zapísaných/prečítaných paketov aj na úrovni samotných rozhraní *MFB* a *MVB*.

V smere *RX* sa rozhraním *MFB* prenášajú samotné dáta paketu, rozhraním *MVB* informácie o pakete – dĺžka paketu, jeho metadáta, číslo kanála, pre ktorý je paket určený a príznak zahodenia. Pakety by mali byť generované o náhodnej dĺžke 60 – 8 192 bajtov.

Dĺžka by mala byť generovaná v niekoľkých režimoch tak, aby sa často generovali pakety s minimálnou a maximálnou dĺžkou v rade za sebou. Číslo kanála by sa malo generovať náhodne. Čísla kanála by sa mali generovať s približne uniformnou pravdepodobnosťou, mali by sa generovať aj v režime zhlukov, teda aby sa vygeneroval rovnaký kanál pre niekoľko paketov za sebou. Prostredie by preto malo explicitne zbierať informáciu o počte vygenerovaných paketov za sebou pre jeden kanál rovnako, ako aj celkový počet vygenerovaných paketov. Ak má paket nastavený príznak *Discard*, mal by sa zahodiť, inak by sa mal zapísať do pamäte. Tento príznak by sa mal nastavovať pre 20% paketov, mal by sa tiež nastavovať v režime zhlukovania, teda sa bude náhodne nastavovať do log.0 a log.1 pre viac paketov po sebe. Ďalším parametrom generovania je, ako často sa majú nové pakety vygenerovať. V prípade generického parametra *Blocking* mode v log.1 platí, že v prípade preplnenia vnútorných front by *DMA* modul mal prestať prijímať ďalšie pakety nastavením *DST_RDY* do log.0 na vstupných rozhraniach. Takáto situácia by počas simulácie mala nastať. Pakety by sa tiež mali zapisovať na rozhrania v niekoľkých režimoch, kedy sa nezapisujú žiadne/len málo paketov, kedy sa zapisujú s náhodnými medzerami a kedy sa zapíše nový paket hneď po prijatí predošlého.

Na rozhrania v smere *TX* zapisuje *DMA* modul pakety na odoslanie. Na rozhranie *MVB* zapisuje metadáta paketu, jeho dĺžku a číslo kanála, ktorý paket odoslal. Požiadavky na veľkosť odoslaných paketov sú popísané v časti týkajúcej sa softvérových ovládačov. Prostredie by malo pokryť počet paketov odoslaných jednotlivými kanálmi, počet by mal byť približne rovnaký. Mal by tiež pokryť počet paketov, ktoré sa na rozhranie zapísali zaradom z jedného kanála bez toho, aby boli prerušené paketom iného kanála. Ich počet by mal byť dostatočne pokrytý náhodným spúšťaním a zastavovaním softvérových ovládačov pre jednotlivé kanály spolu s náhodnými latenciami medzi pripravovaním nových deskriptorov. U rozhraní je potrebné vhodne nastavovať hodnotu *DST_RDY*. Hodnoty log.0 a log.1 v režime zhlukov. Takéto generovanie by potenciálne mohlo viesť k otestovaniu rozličného zaplnenia výstupnej vyrovnávacej pamäte *DMA* modulu. Do tejto pamäte sa zapisujú pakety zo všetkých kanálov v rovnakom poradí, ako sú zapísané na výstup.

Na úrovní rozhraní *MVB* a *MFB* je potrebné pokryť v každom smere pokrytie hodnôt portov *SRC_RDY* a *DST_RDY*. Tiež by sa mal vždy pokryť počet taktov, počas ktorého tieto porty zotrvali nastavené v rovnakej logickej úrovni. Medzi jednotlivými prenesenými paketmi by sa mali pokryť medzery v rámci daných slov aj počet taktov medzi validnými slovami. Zbernica *MVB* môže preniesť niekoľko položiek za takt, pričom položky môžu byť v slove poskladané s rozličnými medzerami. Preto by sa mali pokryť všetky kombinácie počtu a rozloženia jednotlivých položiek. Tieto pozície by mali byť plne pokryteľné v smere *RX*. Pokrytie v smere *TX* môže byť ovplyvnené spôsobom implementácie zapisovania paketov na toto rozhranie. Na rozhraní *MFB* možno v prípade krátkych rámcov pokryť ich počet prenesený v jednom slove. Možno tiež pokryť rozličné kombinácie začiatkov a koncov rámcov. Pokryť by sa mali aj všetky hodnoty začiatkov a koncov rámcov. Pokrytie pozícií začiatkov rámcov v smere *TX* bude závislé na implementácií zápisu na toto rozhranie. V každom smere možno pokryť počet hlavičiek, ktoré boli zapísané bez zapísania príslušných dát paketov. V smere *RX* možno toto pokrytie ovplyvniť náhodným generovaním medzier medzi vkladanými slovami. V smere *TX* vhodným nastavovaním portov *DST_RDY*.

6.3 Kontrolované vlastnosti

Táto podkapitola sa venuje vlastnostiam, ktoré treba kontrolovať pri verifikácii. Verifikačné prostredie by malo obsahovať mechanizmy kontrolujúce prípadné porušenie vymenovaných

vlastností. Podkapitola sa venuje rozhraniam k sieťovej komponente, komunikácii so softvérovým ovládačom a správnemu prístupu do pamäte.

Pri odosielaní paketov v smere *TX* by sa pre každý kanál malo kontrolovať, že boli odoslané všetky pakety obsiahnuté v nachystaných deskriptoroch. Pri prenesení by sa nemala meniť dĺžka ani obsah paketu. Kanály fungujú nezávisle, platí však, že jeden kanál by mal odosielať pakety v rovnakom poradí, aké bolo poradie deskriptorov. Deskriptor obsahuje navyše metadáta, ktoré je potrebné preniesť spolu s paketom. Metadáta, dĺžka paketu a číslo kanála, ktorý paket odoslal, sa prenášajú prostredníctvom rozhrania *TX MVB* ku sieťovej komponente. Prenesené položky *MVB* sa párujú s paketmi odoslanými rozhraním *TX MFB* pomocou rovnakého poradia. Pri párovaní by sa malo kontrolovať, že dĺžka rámca preneseného rozhraním *MFB* sa zhoduje s dĺžkou paketu v hlavičke. Kanál v smere *TX* začne pakety odosielať po spustení, keď od softvéru obdrží deskriptory s paketmi. Potom, ako softvér pripraví deskriptory na odoslanie všetkých paketov, môže vyžiadať zastavenie daného kanála. Kanál pred zastavením ešte odošle všetky pakety z nachystaných deskriptorov. V prípade, že softvér nesprístupní všetky deskriptory pre posledný paket, môže dôjsť k zaseknutiu. Toto však indikuje nesprávny prístup zo strany softvéru. Preto bude ovládač zapisovať len kompletné pakety. Pri práci s deskriptormi softvérový ovládač najskôr alokuje pamäť s paketmi a nachystá deskriptory, ktoré zapíše do vyrovnávacej pamäte. Následne aktualizuje softvérový ukazateľ. Modul *DMA* signalizuje, že dané deskriptory použil, nastavením hardvérového ukazateľa. Po aktualizácií hardvérového ukazateľa môže softvér ľubovoľne upravovať obsah pamäte odkazovanej použitými deskriptormi, prípadne ju uvoľniť. Preto by *DMA* modul už nemal pristupovať do odkazovanej pamäte po aktualizácií hardvérového ukazateľa. Túto vlastnosť je dôležité kontrolovať najmä v kombinácií s náhodným predbiehaním čítacích požiadavkov zápisovými tak, ako to umožňuje špecifikácia zbernice *PCIe*.

DMA modul tiež prijíma pakety od sieťovej karty. Sieťová karta určuje, ktorým kanálom sa má daný paket spracovať. Kanály pre prijímanie sú staticky priradené koncovým bodom *DMA*, ktoré ich obsluhujú. Softvér sa stará o spúšťanie a zastavovanie kanálov. Ak je kanál zastavený, paket sa musí zahodiť. Po spustení kanála softvér pomocou deskriptorov definuje, kam sa prijaté pakety uložia. Pakety sa potom začnú zapisovať do pamäte odkazovanej aktuálnymi deskriptormi. Pakety sa v rámci jedného kanála ukladajú v poradí, v ktorom boli prijaté. Môže sa stať, že sa paket nezmestí do aktuálneho deskriptoru a daný deskriptor nepovoľuje zapísanie paketu do viac deskriptorov. V takom prípade sa paket nemôže začať zapisovať a musí sa zahodiť. U každého prijatého paketu je do pamäte zapísaná jeho hlavička. Podobne, ako v smere *RX* platí, že *DMA* modul by už nemal pristupovať do pamäte odkázanej deskriptormi, ktoré už uvoľnil aktualizovaním hardvérového ukazateľa.

V časti 5.2 sa spomínalo, že pôvodné prostredie určovalo, či sa má paket zapísať, použitím mikroarchitekturálneho rozhrania. Vďaka tomu bolo možné overiť, nasledujúce vlastnosti:

- ak bol paket určený sieťovou kartou na zahodenie, skutočne sa *DMA* modulom zahodil,
- ak bol paket určený *DMA* modulom na zápis do pamäte, skutočne sa zapísal,
- ak bol paket určený *DMA* modulom na zahodenie z iných dôvodov, bol zahodený.

Tento spôsob kontroly by bolo možné potenciálne rozšíriť pridaním ďalších informácií o aktuálnych prenosoch. Platí, že po spustení kanála sa prijaté pakety prestanú zahadzovať a začnú sa zapisovať do pamäte. Kontrolný mechanizmus by teda mohol kontrolovať, že

odkedy sa prvý paket po spustení zapíše do pamäte, už žiadny paket prijatý neskôr sa nezahodí až do vypnutia kanála. Ďalej by mohlo byť možné bližšie overiť príčinu zahodenia paketu. Paket by sa pri spustenom kanáli mal zahodiť len v týchto prípadoch:

- sieťovou kartou bol nastavený bit *Discard* pre daný paket,
- generický parameter *Blocking Mode* bol nastavený do log.0 a došlo preplneniu vnútornej vyrovnávacej pamäte,
- paket sa nezmestí do aktuálneho deskriptoru.

Situáciu, kedy došlo k preplneniu vnútorných front, by bolo možné kontrolovať monitorovaním vnútorných signálov. Situáciu, kedy sa paket nezmestí do aktuálneho deskriptoru, by bolo možné overiť ukladaním presnej informácie o deskriptoroch, do ktorých sa daný paket má zapísať.

Pri prístupe do pamäte by požiadavky nemali prekračovať hranicu stránky. Veľkosť požiadavku na čítanie by nemala prekročiť veľkosť danú parametrom *MRRS*, požiadavku na zápis veľkosť *MPS*. Prostredie by malo naviac kontrolovať, že softvérové ovládače pristupujú len do pamäte hlavičiek, deskriptorov, do vyrovnávacej pamäte pre aktualizáciu hardvérových ukazateľov a pamäte odkázanej aktuálnymi deskriptormi. Každý požiadavok na čítanie by mal mať unikátnu hodnotu *Tag* v rámci požiadavkov, ktoré sa odbavujú. Hodnota *Tag* musí byť jedinečná pre jeden identifikátor jednotky (*UnitID*).

Koniec testu by mal byť indikovaný zastavením všetkých kanálov. Prípadné nezastavenie kanála môže indikovať jeho zaseknutie.

Kapitola 7

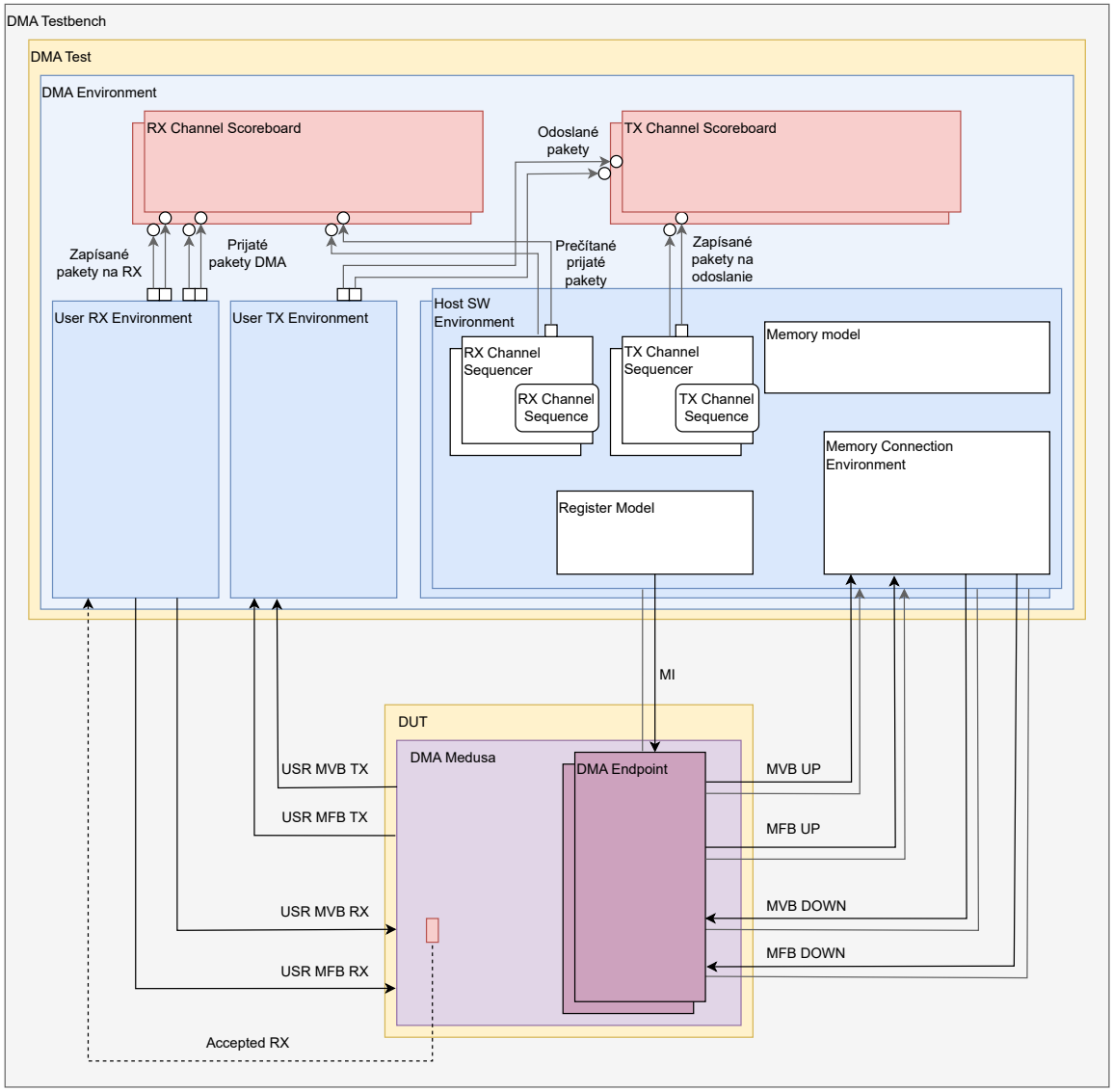
Vytvorenie verifikačného prostredia podľa metodiky UVM

Táto kapitola sa venuje návrhu a implementácií verifikačného prostredia podľa metodiky UVM. Prostredie vychádza z funkčných požiadavkov definovaných v kapitole 6. V úvode kapitoly sú ukázané hlavné komponenty prostredia, ďalšie časti obsahujú ich podrobný popis vrátane kľúčových algoritmov. V závere je implementácia zhodnotená.

Nové prostredie má komunikovať pomocou rovnakých princípov ako pôvodné. Preto je štruktúra nového prostredia podobná. V pôvodnom prostredí boli použité verifikačné modely rozhraní *PCIe RQ* a *PCIe RC* pre každý modul *PTC*. Prostredníctvom nich mohol systém *DMA* pristupovať do pamäte. Pamäť bola zdieľaná pre všetky koncové body. Nové prostredie nepoužíva moduly *PTC*. Z tohto dôvodu je potrebné pôvodné modely rozhraní nahradiť novými modelmi rozhraní *MVB* a *MFB* pre komunikačné kanály *Up* a *Down*. Tieto modely budú napojené na každý koncový bod *DMA*. Koncové body medzi sebou nijak nezdielajú logiku, ktorá sa týka adresového priestoru pamätí. Každý z koncových bodov má vlastný priestor adries, na ktoré môže pristupovať. Priestor je daný adresami kruhových vyrovnávacích pamätí pre deskriptory, hlavičky a pamäťou odkazovanou deskriptormi. Ak by sa v reálnom hardvéri stalo, že dva koncové body pristupujú do rovnakej pamäte, žiadny z konceptov v systéme *DMA* nezabezpečí, aby tieto prístupy boli koherentné. Preto možno použiť nezávislý model pamäte pre každý koncový bod *DMA*. Takýto postup síce oddiali verifikačné prostredie od reálneho hardvéru, kde koncové body môžu pristupovať do jedného pamäťového priestoru, no zjednoduší a potenciálne urýchli prístupy do pamäte v priebehu testu. Z pohľadu kontrolných mechanizmov sa tiež nič nezmení. Model pamäte je typicky implementovaný ako zoznam polí bajtov. Ovládač každého kanála v modeli nezávisle alokuje, zapisuje a číta dáta. Každý koncový bod môže čítať a zapisovať dáta len do pamäťových blokov alokovaných pre jeho kanály. Ak sa v pamäťovom modeli bude nachádzať menej položiek, pri niektorých implementáciách by mohol byť tento prístup do pamäte potenciálne rýchlejší. Verifikačné prostredie na najvyššej úrovni zapuzdruje prostredia pre každý koncový bod *DMA*. Prostredie každého koncového bodu obsahuje vlastný model pamäte, modely rozhraní pre prijímanie pamäťových požiadavkov a odosielanie odpovedí, model rozhrania *MI* a softvérové ovládače pre všetky kanály, ktoré koncový bod spravuje. Softvérové ovládače pre kanály bežia konkurentne.

Prostredie využíva existujúce verifikačné komponenty. Komponenty boli navrhnuté s cieľom poskytnúť úroveň abstrakcie pri ovládaní a analýze komunikácie na rozhraniach *MVB*, *MFB* a *MI*. Komponenty rozhraní *MVB* a *MFB* sú použité pri konštruovaní prostredí

s vrstvením sekvencií. Rozhranie *MI* slúži pre ovládanie registrov. Komponenty rozhrania *MI* sa používajú pri definovaní komunikácie s registrami na abstraktnej vrstve (*Register Abstraction Layer*).



Obr. 7.1: Základná štruktúra vytvoreného verifikačného prostredia.

Diagram 7.1 ilustruje základnú štruktúru vytvoreného verifikačného prostredia a jeho napojenie na systém *DMA*. Hlavným modulom je *DMA Testbench*. Tento modul bude inštancovať modul *DUT* a triedu *DMA Test*. Trieda *DMA Test* obsahuje prostredia pre rozhrania hostiteľskej pamäte aj sieťovej karty. Na strane hostiteľskej pamäťe sa nachádza prostredie *Host SW* pre každý koncový bod *DMA*. Prostredie *User TX* slúži pre čítanie zaslaných paketov *DMA* modulom cez komunikačný kanál *TX*. Prostredie *User RX* generuje a zasiela pakety, ktoré *DMA* modul zapisuje do pamäte. Prostredie tiež analyzuje pakety zapísané na mikroarchitekturálne rozhranie *Accepted MVB*. Toto rozhranie obsahuje informácie, či sa prijaté pakety zahodia, alebo budú zapísané do pamäte. Trieda *DMA Test* obsahuje aj komponentu *Scoreboard*, ktorá vyhodnocuje správne prenášanie paketov. Kom-

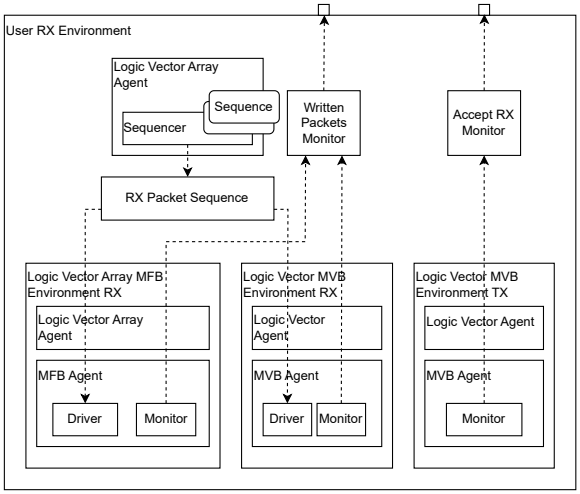
ponenta je napojená na ostatné prostredia, ktoré jej posielajú informácie o odoslaných a prijatých paketoch.

V module *DMA Testbench* sa generujú hodnoty hodinového signálu a hodnota *Reset*. Hodnota *Reset* je generovaná len na začiatku testu. Konkrétny test sa inštancuje za behu, aby nebolo nutné prekladať prostredie pre každý test. Sú tu tiež inštancované rozhrania, referencia na ne sa predáva verifikačným komponentám prostredníctvom *UVM* databázy. Modul *DUT* obsahuje hlavný modul systému *DMA Medusa*. Rozhrania *User RX MVB* a *User TX MVB* prenášajú informácie o paketoch štruktúrovane, teda každá položka sa prenáša osobitným signálom. Tieto informácie je potrebné serializovať do jedného signálu, aby pre ich prenos bolo možné využiť implementované *UVM* prostredia pre rozhrania *MVB* a *MFB*. Serializácia je teda vykonaná v module *DUT*.

Prostredie *User RX* slúži na generovanie paketov, ktoré sa zapisujú na rozhrania od sieťovej karty v smere *RX*. Prostredie ďalej monitoruje zapísané pakety na rozhranie a informáciu, či sa pakety prijali, alebo zahodili. Zapísané pakety sa spolu s informáciou o ich prijatí posielajú do komponenty *Scoreboard*. Pre každý paket je vygenerované číslo kanála, metadáta a samotné dáta paketu o náhodnej dĺžke. Metadáta sa generujú náhodne bez obmedzujúcich podmienok. Číslo kanála sa generuje v náhodnom režime a režime, kedy sa vygeneruje niekoľkokrát za sebou rovnaké číslo kanála. Režim rovnakého čísla kanála sa použije s pravdepodobnosťou 5%, pričom sa vygeneruje 1-20 paketov v rade. Pre generovanie paketových dát je použitý existujuci agent *Logic Vector Array*. Tento agent umožňuje generovanie paketov nasledujúcimi sekvenciami:

- náhodná sekvencia,
- sekvencia, kde sa dĺžka paketu generuje podľa gausovského rozloženia,
- sekvencie, kde dĺžka paketu postupne rastie/klesá,
- sekvencia, kde sa pakety generujú s konštantnou dĺžkou, ktorá sa vygeneruje na začiatku sekvencie.

Sekvencie sa náhodne vyberajú a spúšťajú v rámci knižnice sekvencií. Maximálna dĺžka sekvencie je obmedzená na 10 paketov, veľkosť paketu je nastavená na 60 – 8 192 bajtov. Pakety sa generujú v komponente *User RX Generator*. Komponenta generované pakety číta prostredníctvom portu *Sequence Item Pull Port*. Na základe vygenerovaných dát nachystá a zapíše dáta spolu s hlavičkou do položiek sekvencie pre prostredia *Logic Vector Array MFB RX* a *Logic Vector MVB RX*. Tieto prostredia zapisujú hlavičku a dáta na rozhrania systému. Generované hlavičky a dáta sú zapisované do schránok. Sekvencie bežiace na komponentách Sequencer v prostrediach *Logic Vector Array MFB RX* a *Logic Vector MVB RX* v slučke čítajú položky sekvencie zo schránky a posielajú ich do sekvencií nižšej úrovne. Sekvencie nižšej úrovne z týchto transakcií skladajú dátové slová pre rozhrania *MFB* a *MVB*, pričom vkladajú náhodné medzery medzi rámcami a latencie medzi slovami. Pri analýze komunikácie sú monitorované transakcie na každom rozhraní. Transakcie sú zasielané vysokoúrovňovému monitoru *Written Packets*. Tento monitor z prijatých nízkoúrovňových transakcií skladá vysokoúrovňovú transakciu, ktorá obsahuje dáta paketu, metadáta, príznak zahodenia a kanál, ktorý má paket spracovať. Vysokoúrovňová transakcia sa posiela do komponenty Scoreboard. V prípade mikroarchitekturálneho rozhrania sa posielajú položky obsahujúce číslo kanála a príznak, či sa paket zahodil, alebo bude zapísaný do pamäte. O posielanie položiek sa stará trieda *Accept RX Monitor*. Diagram 7.2 popisuje základnú štruktúru prostredia *User RX*.



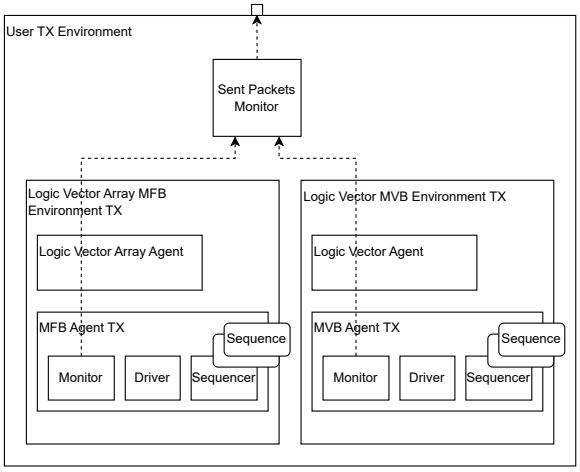
Obr. 7.2: Komponenty prostredia *User RX*.

Prostredie *User TX* slúži pre čítanie paketov, ktoré boli odoslané z modulu *DMA*. Pakety sú prijímané na rozhraniach *MVB* a *MFB*. Prostredie náhodne generuje hodnoty pre porty *Destination Ready*. Jeho štruktúra je podobná prostrediu *User RX*. Rovnako sa tu využíva vrstvenie agentov. V tomto prípade sa však nevrstvia sekvencie, ale len monitorovanie transakcií. Prostredie *Logic Vector Array MFB TX* monitoruje rozhranie *MFB* a posiela prečítané rámce do monitoru na vyššej úrovni. Prostredie *Logic Vector MVB TX* monitoruje rozhranie *MVB* a posiela prečítané položky. Tieto položky sa v monitore *Sent Packets Monitor* zložia do transakcie, ktorá bude obsahovať odoslaný paket, metadáta a číslo kanála, ktorý paket odoslal. V prípade, že sa dĺžka dátového rámca *MFB* nezhoduje s dĺžkou paketu v hlavičke, simulácia sa ukončí s príslušnou chybou. Prijatý paket sa odošle do komponenty *Scoreboard* daného kanála. Monitor za týmto účelom obsahuje pole analytických externých portov, ktoré sú napojené na porty komponent *Scoreboard* jednotlivých kanálov. Hodnota portov *Destination Ready* sa pre rozhrania náhodne generuje v nízkoúrovňových agentoch. Pre generovanie sa používajú rozličné sekvencie. Tieto sekvencie sa náhodne spúšťajú knižnicou sekvencií. Diagram 7.3 popisuje základnú štruktúru prostredia *User TX*.

Prostredie *Host SW* sa stará o komunikáciu s koncovým bodom *DMA*. Prostredie obsahuje sekvencie pre softvérový ovládač každého kanála v oboch smeroch. Tieto sekvencie konfigurujú kanály pomocou rozhrania *MI* a pracujú s pamäťovým modelom. Prostredie obsahuje vnorené prostredie *Memory Connection*, ktoré obsluhuje požiadavky koncového bodu *DMA* pre prístup do pamäte.

Prostredie pracuje s registrami na abstraktnej vrstve. Pri implementácií je použitý existujúci registrový model. Tento registrový model používa triedy pre komunikáciu s rozhraním *MI*. Danému modelu sú definované konkrétne registre systému *DMA*. Registre sú usporiadané v bloku registrov. Každý blok zodpovedá jednému kanálu. Z registrov je vytvorená registrová mapa, ktorá je priradená registrovému modelu. Následne sa k registrom pristupuje pomocou metód *read()* a *write()*.

Pamäťový model je implementovaný podobne ako v predošlom verifikačnom prostredí. Implementovaný je v komponente *Memory Model*. Obsahuje metódy pre alokáciu, zápis, čítanie a uvoľňovanie. Tieto metódy používajú softvérové ovládače kanálov a prostredie *Memory Connection*, ktoré obsluhuje požiadavky na prístup do pamäte modulu *DMA*. Pamäť



Obr. 7.3: Komponenty prostredia *User TX*.

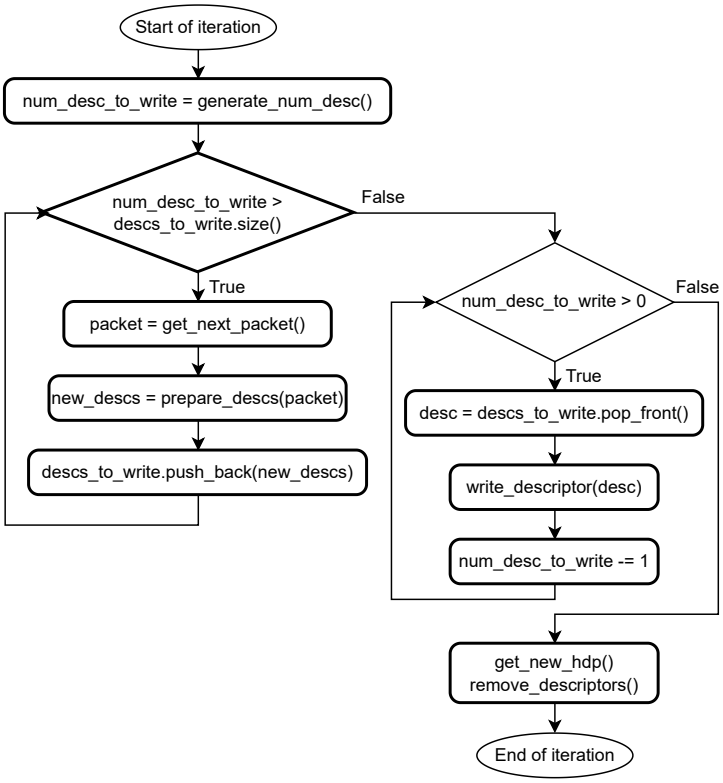
je organizovaná ako dynamické pole pamäťových blokov. Pamäťový blok je implementovaný v triede *Memory Block*. Blok obsahuje počiatočnú adresu, veľkosť bloku, alokované pole pre dáta, adresu poslednej položky v bloku a počet deskriptorov, ktoré na tento blok odkazujú. Metóda *allocate_block()* implementuje alokáciu nového bloku. Argumentom je veľkosť bloku a zarovnanie, návratovou hodnotou je adresa začiatku bloku. Metóda náhodne určí adresu, aby vyhovovala zarovnaniu a nový blok sa neprekrýval so žiadnym existujúcim. Pre čítanie sa používajú metódy *read()* a *read_dwords()*. Metóda read číta dáta po bajtoch. Pred čítaním sa najskôr sekvenčne prejde pole a určí sa blok, z ktorého sa bude čítať. V prípade jeho nenájdenia alebo veľkosti požiadavku presahujúceho koniec bloku sa simulácia zastaví na chybe. Metóda *read_dwords()* číta dáta po 32-bitových slovách. Parametrami sú aj nevalidné bity na začiatku a konci slova žiadaných dát slúžiace pre zarovnanie, hodnoty týchto bajtov sú generované náhodne. Model implementuje analogicky metódy *write()* a *write_dwords()*. Naviac sú implementované metódy pre inkrementovanie a dekrementovanie počítadla deskriptorov, ktoré na pamäťový blok odkazujú. V prípade, že počet referencií po dekrementovaní klesne na hodnotu 0, pamäťový blok je zmazaný.

Sekvencie *TX/RX Channel Sequence* implementujú softvérové ovládače kanálov. Pri implementácií je použité stavové riadenie rovnako ako u predošlej verifikácie:

- v stave *Stopped* je kanál zastavený,
- v stave *Starting* sa kanál nainicializuje,
- v stave *Running* kanál spravuje pamäťové prenosy,
- v stave *Stopping* sa kanál zastaví.

Ovládač kanála v stave *Stopped* zotrvá náhodnú dobu. V stave *Starting* inicializuje registre a alokuje vyrovnávacie pamäte. v Stave *Running* prenesie vygenerovaný počet paketov. V stave *Stopping* dokončí prenosy a zastaví kanál. Prechody medzi stavmi sú implementované v základnej sekvencií *Channel Sequence Base*. Sekvencie pre každý smer potom implementujú metódy, ktoré sa vykonávajú v každom stave.

Pri spustení kanála v smere *TX* sa nainicializujú potrebné registre, alokuje sa pamäť pre deskriptory a hardvérové ukazatele. Veľkosť vyrovnávacej pamäte deskriptorov sa určuje náhodne vo viacerých režimoch, pričom vyššia váha je nastavená na režimy generujúce

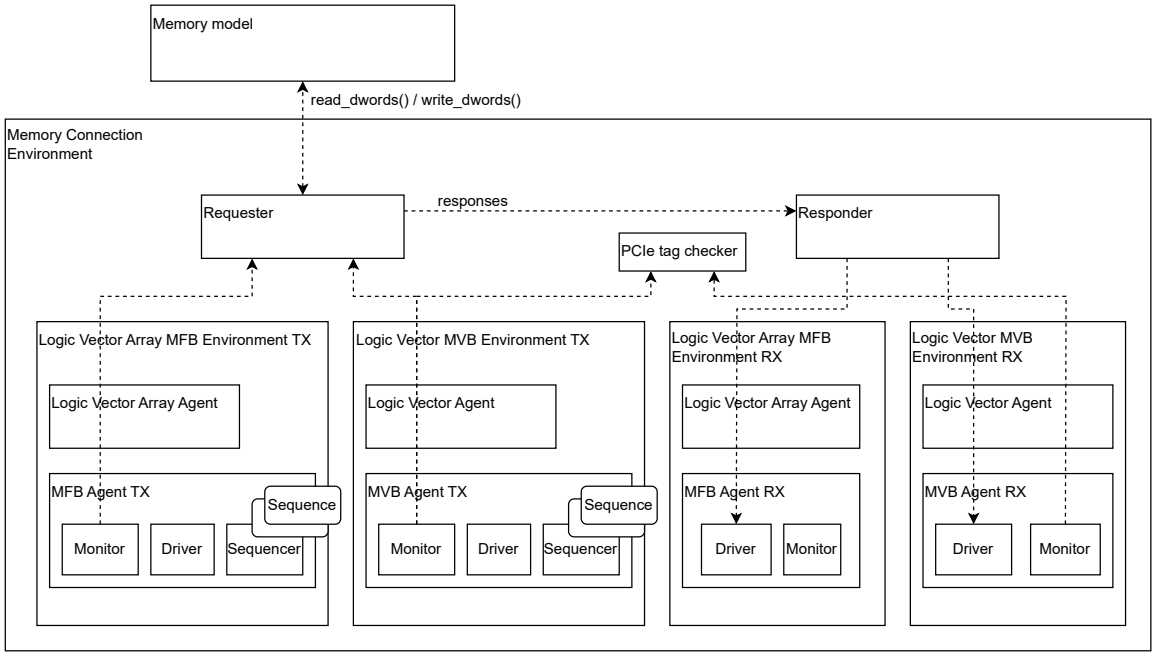


Obr. 7.4: Diagram priebehu jednej iterácie zápisu paketov v smere TX.

nižšie veľkosti. Maximálna veľkosť je 1024 položiek, minimálna 32. Po inicializácií registrov sa spustí kanál zápisom do registru *Control* a počká sa na nastavenie registra Status. Po spustení sa vygeneruje počet paketov, ktoré sa prenesú, kým sa kanál vypne. Minimálna hodnota je 0, maximálna je daná počtom paketov, ktoré sa majú preniesť v rámci testu. Následne sa v cykle pripravujú deskriptory s paketmi, pripravené deskriptory sa zapisujú do pamäte a použité deskriptory sa uvoľnia. Diagram 7.4 zobrazuje priebeh jednej iterácie pri zapnutom kanáli. Medzi jednotlivými iteráciami sa počká 0 – 64 taktov. Pri zápise paketov môže nastať niekoľko možností – niekoľko paketov sa zapíše do jedného pamäťového bloku, alebo sa jeden paket zapíše do niekoľkých pamäťových blokov. V takom prípade sa paket zapíše do niekoľkých deskriptorov, pričom sa v každom deskriptore okrem posledného nastaví príznak *Next* do log.1. Aby bolo možné každú z možností dostatočne otestovať, veľkosť bloku je generovaná nezávisle na veľkosti paketu. Ovládač pracuje tak, že najskôr vygeneruje počet deskriptorov, ktoré pripraví v aktuálnej iterácii. Minimálny počet je 0, maximálny sa určí podľa počtu voľných položiek vo vyrovnávacej pamäti. Pri generovaní sa nastavuje vyššia váha pre nižší počet deskriptorov. Následne sa v cykle pripravujú nové deskriptory s paketmi, kým sa neprekročí daný počet. Je pri tom žiadané, že ak je do posledných deskriptorov zapísaný paket s použitím príznaku *Next*, tak sa tieto deskriptory nemusia zapísať do pamäte v aktuálnom kroku. Ak sa pre paket použije dostatočný počet deskriptorov, bude to znamenať, že sa paket zapíše v niekoľkých iteráciach. Pripravené pakety sa vkladajú do frontu *descs_to_write*. Zapísané pakety sa odosielajú aj do komponenty *Scoreboard*. Po pripravení dostatočného počtu deskriptorov sa deskriptory zapíšu do pamäte a aktualizuje sa softvérový ukazateľ. Následne sa prečíta aktuálna hodnota hardvérového ukazateľa a pamäť odkazovaná použitými deskriptormi sa uvoľní. Keď sa pripraví počet

paketov, ktorý sa vygeneroval pri spustení kanála, prejde sa do stavu *Stopping*. V tomto stave sa dokončí zápis pripravených deskriptorov do vyrovnávacej pamäte a počká sa, kým modul *DMA* neprečíta všetky deskriptory. Následne sa vyžiada ukončenie kanála zápisom do registra *Control* a počká sa, kým sa kanál zastaví. Po zastavení sa uvoľní pamäť pre hardvérové ukazatele a deskriptory.

Ovládač v smere *RX* pracuje obdobne k ovládaču pre *TX*. Pri inicializácií prenosu naviac alokuje vyrovnávaciu pamäť hlavičiek. Pamäť hlavičiek má veľkosť 32 – 2 048 položiek. Pri spustení sa tiež vygeneruje počet paketov, ktoré sa príjmu v danom behu. Po spustení v cykle pripravuje nové deskriptory, číta dáta a hlavičky prijatých paketov a aktualizuje softvérové ukazatele. V každej iterácií alokuje náhodný počet deskriptorov, ktoré pripraví. Počet deskriptorov bude 0 až počet voľných miest vo vyrovnávacej pamäti. Počet sa generuje v niekoľkých režimoch, pričom najvyššia váha je na nízkom počte deskriptorov (0 – 20% voľných položiek). Alokované bloky majú veľkosť 60 – 8 192 bajtov. Ak sa horných 34 bitov adresy bloku zhoduje s predchádzajúcim blokom, konfiguračný deskriptor sa nepoužije. Po pripravení deskriptorov ovládač získa aktuálnu hodnotu hardvérového ukazateľa do pamäte deskriptorov a prečíta prijaté pakety. Platí, že jeden paket môže byť zapísaný v niekoľkých deskriptoroch a zároveň nie je požadované žiadne poradie medzi zapísaním hlavičky a dát do deskriptorov. Ovládač prečíta uvoľnené deskriptory a odstráni odkazované pamäťové bloky z pamäte. Pamäťové bloky si uloží do interného frontu. Následne si prečíta hlavičky. Hlavičky si uloží do tiež do vnútorného frontu. Potom z pamäťových blokov extrahuje kompletné pakety. Pri tom najskôr zistí veľkosť paketu v prvej hlavičke. Ak má k dispozícií dostatočný počet pamäťových blokov, prečíta paket, pamäťové bloky uvoľní a hlavičku odstráni z frontu. Prečítaný paket pošle do komponenty *Scoreboard*. Po prijatí požadovaného počtu paketov sa požiada o zastavenie kanála zápisom do registra *Control*. Počas zastavovania môže ešte kanál dokončiť príjem rozpracovaných paketov, preto sa pokračuje v pripravovaní nových deskriptorov a čítaní paketov až do zastavenia kanála. Po zastavení sa uvoľní použitá pamäť.



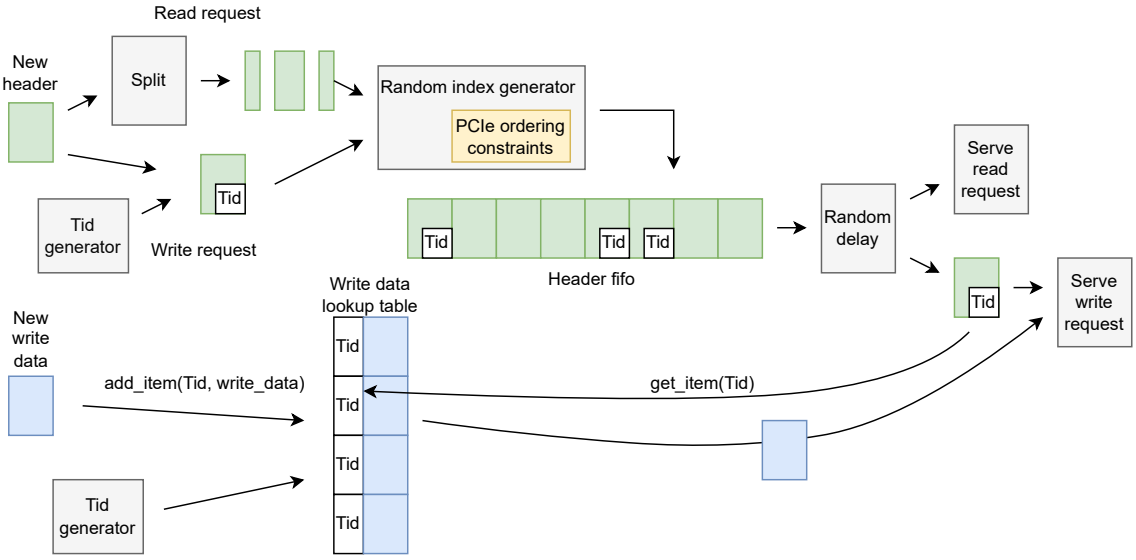
Obr. 7.5: Komponenty prostredia *Memory Connection*.

Prostredie *Memory Connection* zapuzdruje rozhrania pre odbavenie pamäťových požiadavkov koncového bodu *DMA*. Diagram 7.5 zobrazuje komponenty tohto prostredia. Pre prijímanie požiadavkov do pamäte sa starajú prostredia *Logic Vector Array MFB TX* a *Logic Vector MVB TX*. Tieto prostredia posielajú prečítané transakcie do analytickej komponenty *Requester*. Komponenta *Requester* odbavuje požiadavky v náhodnom poradí. Po odbavení požiadavkov na čítanie odošle prečítané dáta komponente *Responder*, ktorá na základe odpovede vytvorí hlavičku a dáta pre rozhrania *MVB* a *MFB Down*. Pre zápis odpovedí sú použité existujúce prostredia *Logic Vector Array MFB RX* a *Logic Vector MVB RX*. Komponenta *Requester* odbavuje požiadavky v náhodnom poradí tak, ako je to definované v podkapitole 6.2. Dôležité je pri tom nielen generovať náhodné poradie odpovedí na čítanie, ale aj predbiehanie požiadavkov pri ich vykonaní. Komponenta obsahuje front, do ktorého sa vkladajú prijaté hlavičky. Po vybratí hlavičky z frontu sa požiadavok odbaví a v prípade požiadavku na čítanie sa odošle odpoveď. Zbernica *PCIe* formuluje možnosti predbiehania pamäťových požiadavkov tak, že vymedzí, ktoré transakcie môže aktuálne prijatá transakcia predbehnúť. Tieto pravidlá možno implementovať vkladaním hlavičiek na náhodné pozície do frontu tak, aby boli pravidlá dané špecifikáciou dodržané. Hlavičky sa potom vyberajú v poradí. U požiadavkov na čítanie je tiež žiadané, aby boli odpovede rozložené na viac častí. Naviac, pamäťový model generuje aj situáciu, kedy požiadavok na zápis predbehne len niektoré časti prečítaných odpovedí. Toto je implementované tak, že sa požiadavok na čítanie rozloží na viac častí ešte pred jeho vložením do frontu. Pre každú časť sa potom aktualizuje adresa, veľkosť požiadavku a pripíše sa príznak, či prečítané dáta budú predstavovať poslednú časť odpovede. Pozície pre nové požiadavky sa generujú s nasledujúcimi pravidlami:

- pozícia požiadavku na zápis je v prípade príznaku *Relaxed Ordering* v log.0 daná veľkosťou frontu a pozíciou posledného požiadavku na zápis vo fronte (požiadavky na zápis by sa nemali predbiehať),
- pozícia požiadavku na čítanie je daná veľkosťou frontu a pozíciou posledného požiadavku na zápis, alebo pozíciou posledného požiadavku s rovnakými položkami *UnitID* a *Tag*. Platí, že požiadavok na čítanie nemôže predbehnúť požiadavok na zápis. Ďalej platí, že v časti jednej odpovede na čítanie sa vracajú v poradí. V prípade zhodných položiek *UnitID* a *Tag* ide o predchádzajúcu časť rovnakého požiadavku na čítanie.

Vyberanie hlavičiek z frontu prebieha v poradí, prostredníctvom metódy *pop_front()*. Pri vyberaní sa cyklicky čaká, kým front bude neprázdny. Následne sa vygeneruje náhodný počet taktov, po ktoré sa počká. Týmto sa simuluje nenulová latencia pamäte. Potom sa vyberie hlavička. V prípade požiadavku na zápis sa k hlavičke priradia dáta. Dáta sa priraďujú na základe rovnakého poradia prijatia na vstupných rozhraniach. Za týmto účelom sú implementované sekvenčné čísla, ktoré sa inkrementujú separátne pre hlavičky aj dáta. Sekvenčné číslo hlavičky sa zapíše do nej pri vkladaní do frontu. Prijaté dáta sa zapisujú do asociatívneho poľa, kde kľúčom je aktuálne sekvenčné číslo. Po priradení dát k hlavičke sa požiadavok odbaví volaním metódy pamäťového modelu *write_dwords()* v prípade zápisu, v prípade čítania *read_dwords()*. Schéma 7.6 ilustruje odbavovanie transakcií. Položky *Tid* predstavujú sekvenčné čísla. Generátor položiek *Tid* inkrementuje sekvenčného číslo pre každý požiadavok.

Úlohou komponenty *Responder* je rozloženie požiadavku na hlavičku a dáta pre rozhrania *MFB* a *MVB*. Komponenta každú prijatú transakciu rozloží na hlavičku a dáta. Hlavička a dáta sa následne posielajú do prostredí príslušných rozhraní prostredníctvom



Obr. 7.6: Schéma generovania náhodného poradia odbavenia *PCIe* transakcií.

schránok. Rozhrania *Logic Vector Array MFB RX* a *Logic Vector MVB RX* implementujú princíp vrstvenia, pričom na komponentách *Sequencer* najvyššej úrovne bežia sekvencie, ktoré vyberajú zo schránok položky sekvencie a posielajú ich do sekvencií nižšej úrovne. Nízkoúrovňové sekvencie skladajú dátové slová s náhodnými medzerami a latenciami medzi slovami. Poskladané slová zapíšu na rozhrania.

Z pohľadu kontrolných mechanizmov sa má pri komunikácii s pamäťou kontrolovať maximálna veľkosť požiadavku na čítanie a zápis, či požiadavok neprekračuje hranicu stránky a jedinečnosť hodnoty *Tag*. Veľkosť a prekročenie hranice stránky sa kontroluje v komponente *Requester* pri prijatí hlavičky požiadavku. Pre kontrolu jedinečnosti hodnoty *Tag* je vytvorená komponenta *PCIe Tag Checker*. Ide o analytickú komponentu, ktorá číta hlavičky aktuálne prijatých požiadavkov a odoslaných odpovedí. Kontroluje sa jedinečnosť kombinácie hodnôt *Tag* a *UnitID* od prijatia hlavičky požiadavku až po odoslanie poslednej časti odpovede. Aby bola kontrola presná v každom takte, hlavičky sa čítajú priamo z analytických portov rozhraní *MVB Up* a *Down*. Komponenta si interne udržuje asociatívne pole požiadavkov. Kľúčom je kombinácia hodnôt *Tag* a *UnitID*. Ak sa pri prijatí nového požiadavku daná kombinácia hodnôt v poli už nachádza, simulácia sa zastaví s príslušnou chybou. Po prečítaní hlavičky poslednej časti odpovede sa položka z poľa odstráni.

Vo verifikačnom prostredí bolo potrebné generovať viacbitové hodnoty v niekoľkých režimoch. Za týmto účelom bola vytvorená trieda *Range Value Gen*. Hlavnou úlohou triedy je generovanie hodnôt vo vopred zadanom intervale, pričom sa u generovania použije viac režimov. Režimy sa majú nedeterministicky prepínať. Takéto generovanie možno použiť napríklad pri generovaní veľkosti pamäťového bloku, počte paketov alebo pri generovaní náhodných latencií. Implementované režimy sú generovanie minimálnej hodnoty, maximálnej hodnoty, náhodný režim, režim zhlukovania a režimy pre generovanie nízkych a vysokých hodnôt. Režim zhlukovania vracia počas trvania rovnakú hodnotu, ktorú si náhodne vygeneroval na jeho začiatku. Režimy nízkych a vysokých hodnôt generujú len 20% najnižších/najvyšších hodnôt. Pri inicializácií generátora sa po jeho vytvorení zadá interval, minimálna a maximálna dĺžka trvania režimu a váhy jednotlivých režimov. Nastavenie váh je vhodné použiť napríklad pri generovaní latencie, kedy je vhodné zvýšiť pravdepodob-

nosť hodnôt blízkych 0. Ďalej je možné vypnúť použitie niektorých režimov v prípade, ak sa žiada generovať hodnoty s uniformnou pravdepodobnosť. Toto sa využíva pri generovaní čísla kanála. Nová vygenerovaná hodnota sa získava metódou *get_next_value()*. Niekedy je potrebné obmedziť minimálnu a maximálnu hodnotu dynamicky, preto je naviac pridaná metóda *get_next_value_in_range()*, ktorá pred získaním ďalšej hodnoty nastaví minimálnu a maximálnu hodnotu. Dynamické nastavenie rozsahu sa používa pri počte deskriptorov, ktoré sa vygenerujú v danom kroku. Hodnota je v tomto prípade obmedzená počtom voľných položiek vo vyrovnávacej pamäti.

Prostredie obsahuje komponenty *Scoreboard* pre každý kanál. V smere *TX* sa porovnáva, či sa paket zapísaný do pamäte softvérovým ovládačom zhoduje s paketom odoslaným modulom *DMA*. Za týmto účelom sú komponenty napojené na monitor odoslaných paketov v smere *TX* a softvérové ovládače kanálov. Porovnáva sa dĺžka paketu, metadáta a jeho obsah. U paketov sa tiež kontroluje ich poradie. V prípade, že sa na rozhraní *User TX* objaví paket, ktorý nebol odoslaný softvérovým ovládačom, simulácia sa ukončí s príslušnou chybou. Na konci simulácie sa overí, či boli odoslané všetky zapísané pakety. Komponenta *Scoreboard* v smere *RX* kontroluje:

- či sa prijaté pakety na rozhraní *User RX* zapísali do deskriptorov správnom poradí,
- či sa pakety určené na zahodenie naozaj zahodia a tie, ktoré sa nezahodili, sa naozaj zapíšu,
- či sa paket zapísaný na rozhranie *User RX* s príznakom *Discard* v log.1 určí vnútornou logikou na zahodenie a naozaj sa zahodí.

Komponenta obsahuje analytické porty, ktoré sú napojené na monitor zapísaných paketov rozhrania *User RX*, na monitor mikroarchitekturálneho rozhrania *Accept MVB* a na softvérový ovládač daného kanála. Po zapísaní vygenerovaného paketu na rozhranie *User RX* sa paket odošle do komponenty *Scoreboard*. Paket sa tu uloží do frontu zapísaných paketov. Na rozhranie *Accept MVB* sa zapíše, či bude paket modulom *DMA* zahodený. Po prijatí transakcie rozhrania *Accept MVB* sa skontroluje, či mal paket príznak *Discard* nastavený v log.0. Ak áno a nezahodil sa, simulácia sa ukončí s príslušnou chybou. Následne sa paket uloží do frontu prijatých paketov. Po prečítaní paketu z pamäte softvérový ovládač odošle paket do komponenty *Scoreboard*, kde sa porovná s prvým paketom vo fronte prijatých paketov. Porovná sa dĺžka paketu, metadáta a jeho obsah. Na konci simulácie sa skontroluje, či sa všetky nezahodené pakety prijali.

7.1 Zhodnotenie implementácie

V tejto kapitole bolo popísané implementované verifikačné prostredie. Cieľom prostredia je zachovať kontrolné mechanizmy a generovanie stimulu, ktoré boli implementované v pôvodnom prostredí. V niektorých prípadoch bol pôvodný stimul nahradený iným, ktorý by mal ekvivalentne otestovať verifikovaný systém. Stalo sa tak v dvoch prípadoch:

- pôvodné prostredie obsahovalo testy s dvomi veľkosťami vyrovnávacej pamäte deskriptorov a hlavičiek. V novom prostredí bol tento mechanizmus nahradený generovaním veľkosti pred každým spustením kanála s tým, že počet položiek bol rozšírený, aby sa vyrovnávacia pamäť nachádzala potenciálne vo viacerých stránkach,

- pôvodné prostredie obsahovalo testy s rôznymi dĺžkami paketov. V novom prostredí sú separátne testy nahradené knižnicou sekvencií, ktorá obsahuje sekvencie s rozličnými dĺžkami paketov, pričom sa sekvencie spúšťajú náhodne.

Rozšírením oproti pôvodnej verifikácii je náhodné poradie odbavovania požiadavkov v pamäťovom modeli podľa pravidiel zbernice *PCIe*. S náhodným poradím odbavovania je možné lepšie otestovať závislosti medzi požiadavkami. Tým je možné odhaliť prípadné chyby, ktoré by potenciálne mohli nastať po napojení systému *DMA* na pamäťový systém poskytnutý treťou stranou. Nové prostredie tiež implementuje kontrolné mechanizmy pôvodného. V podkapitole 6.3 je popísaná možnosť rozšírenia kontrolných mechanizmov o kontrolu dôvodu zahodenia paketu v smere *RX*. Toto rozšírenie by bolo možné potenciálne implementovať po napojení na ďalšie vnútorné signály modulu *DMA* a po rozšírení komponent *Scoreboard* o stavové chovanie. Rozšírenie nie je momentálne súčasťou prostredia.

Prostredie je možné spustiť v rovnakých hardvérových konfiguráciach ako pôvodné. Pôvodné prostredie pri konfigurácii s jedným koncovým bodom *DMA* a ôsmimi kanálmi pre každý smer simulovalo prenos 2000 paketov v každom smere približne 45 minút. Prenesené pakety sa počítali zo všetkých kanálov pre každý smer dohromady. Parametrom nového prostredia je počet paketov, ktoré sa majú preniesť každým kanálom. Ekvivalentom k pôvodnému počtu 2000 paketov je pri danej hardvérovej konfigurácii prenos 250 paketov každým z kanálom. U nového prostredia trvá prenos tohto počtu približne 60 minút. Ide teda o mierne spomalenie voči pôvodnému prostrediu. Vplyv na čas simulácie môže mať latencia pamäťových operácií, ako často sa kanály spúšťajú, zastavujú a koľko paketov prenesú po každej aktivácii kanála. V novom prostredí tiež všetky kanály prenesú rovnaký počet paketov. Ak niektorý kanál vykoná prenos rýchlejšie, na konci simulácie čaká na dobehnutie ostatných kanálov.

Výhodou nového prostredia je vytvorenie zoznamu požiadavkov na generovanie a kontrolované vlastnosti pred jeho implementáciou. S ohľadom na ne bolo možné vhodne navrhnúť prostredie, ktoré tieto požiadavky spĺňa v dostatočnej miere. Na základe požiadavkov bolo tiež možné priradiť prioritu jeho vylepšeniam. Vďaka dopredu definovaným požiadavkom bolo tiež možné navrhnúť funkčné pokrytie. Funkčné pokrytie je rozšírenie oproti pôvodnému prostrediu, jeho vytvoreniu a analýze sa venuje kapitola 8. Poslednou výhodou nového prostredia je samotné použitie metodiky *UVM*. Metodika jasne definuje, akým spôsobom sa majú vytvoriť a napojiť univerzálne verifikačné komponenty. Nové prostredie používa existujúce balíky pre rozhrania *MVB*, *MFB* a existujúci registrový model. Použitie registrového modelu umožnilo vytvorenie registrových máp, ktoré bude v prípade budúcich zmien možné jednoducho aktualizovať. Prístup k registrom bol tiež abstrahovaný od samotného rozhrania *MI*. Prostredie používa sekvencie na rôznych úrovniach. Sekvencie sú spúšťané zo základného testu. Ak sa v budúcnosti vyskytne potreba definovať nový test zameraný na konkrétnu oblasť, bude potenciálne možné vytvoriť ho len samotnou zmenou sekvencií, ktoré sa majú spustiť.

Kapitola 8

Overenie generovaného stimulu pomocou funkčného pokrytia

Táto kapitola sa venuje implementácii funkčného pokrytia, ktorým je možné skontrolovať, či implementované prostredie popísané v kapitole 7 generuje očakávaný stimul. Body pokrytia vychádzajú primárne z požiadavkov na stimul. Požiadavky boli zhrnuté v podkapitole 6.2. Podkapitola sa venuje komunikácii so softvérom, komunikácii s rozhraniami smerom k sieťovej karte a interakcii systému s pamäťou. U funkčného pokrytia som sa zameral najmä na komunikáciu s rozhraniami k sieťovej karte a pamäti. Pre prostredia implementujúce túto komunikáciu som použil existujúce komponenty najmä na napojenie rozhraní *MVB*, *MFB* a na generovanie paketov. Pre pripojenie existujúcich prostredí som použil zložitejšie mechanizmy, ako sú napríklad schránky. V prípade pamäte som zase považoval za pomerne náročné odhadnúť správne parametre generovania poradia odbavenia požiadavkov a ich latencie. Z vymenovaných dôvodov som tak považoval za prioritné monitorovať najmä tieto prostredia. Na záver som vytvoril aj body funkčného pokrytia, ktoré sa zameriavajú na pokrytie rozhraní *MVB* a *MFB* na úrovni vstupných a výstupných portov. Táto skupina bodov pokrytia je použitá u všetkých vonkajších rozhraní *MVB* a *MFB*. Po zmeraní pokrytia som nedostatočne pokrytým bodom priradil prioritu a u bodov s vyššou prioritou som upravil náhodný stimul tak, aby došlo k zlepšeniu pokrytia.

Pokrytie som sa rozhodol zmerať na jednej z verifikovaných konfigurácií, ktorá je dostatočne všeobecná a jej pokrytie by sa malo líšiť od ostatných čo najmenej. Líšiť by sa nemala najmä vnútorná logika a prepojenia medzi vnútornými komponentami. Za týmto účelom som vybral konfiguráciu obsahujúcou:

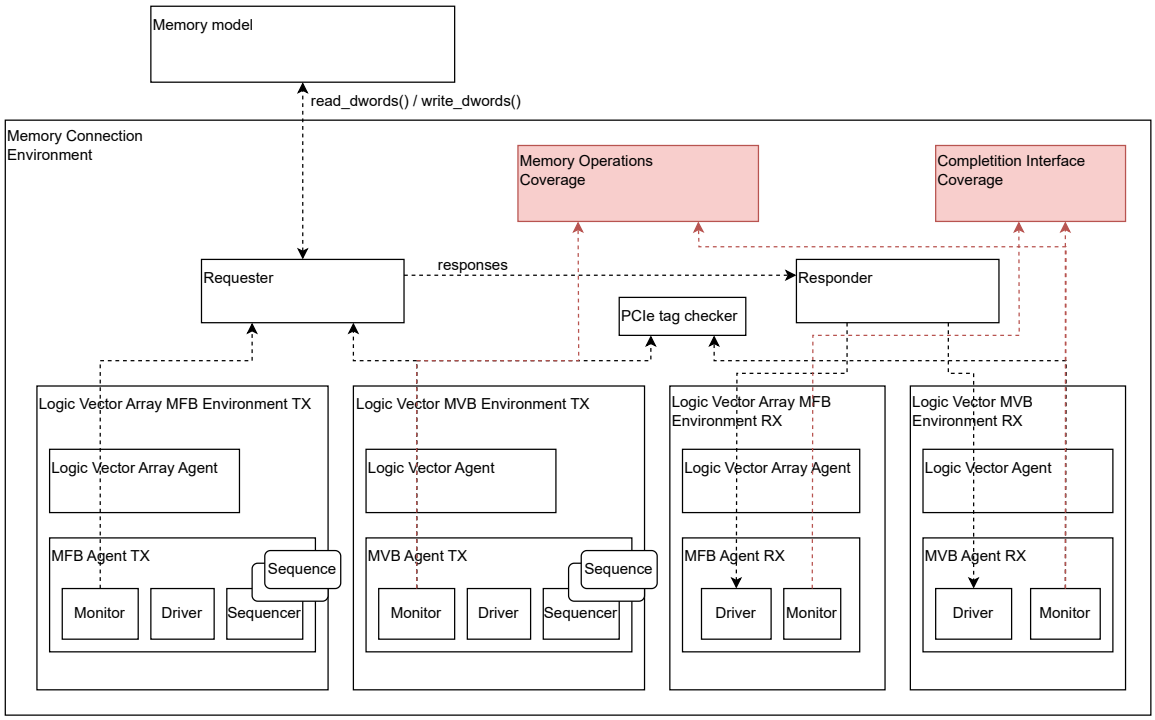
- dva koncové body DMA,
- dva regióny u rozhraní MFB na strane k sieťovej karte,
- zakázaný blokujúci režim,
- 8 kanálov pre každý koncový bod v oboch smeroch.

Pokrytie som meral pri prenesení 250 paketov každým kanálom.

8.1 Pokrytie komunikácie s pamäťou

Dôležitosť pokrytia komunikácie s pamäťou je vyjadrená najmä v podkapitole 6.1. Spomenuté bolo, že pokrytie musí byť dostatočné nielen z pohľadu jednotlivých portov rozhraní,

ale aj rôznych časových závislostí a počtu pamäťových požiadavkov, ktoré sa predbehnú pri ich odbavení. Pri implementácií pokrytia som sa zameral najmä na počet požiadavkov, ktoré čakajú na odpovede, na počet taktov medzi zaslaním požiadavku a prijatím odpovede, rôzne parametre odoslaných požiadavkov a vrátených odpovedí. Keďže sa odoslané odpovede delia na hlavičky zapisované rozhraním *MVB* a dátové rámce zapisované rozhraním *MFB*, zaznamenáva sa aj počet zapísaných hlavičiek bez príslušných dátových rámcov a naopak, počet zapísaných rámcov bez príslušných hlavičiek. Aby bolo možné odmerať najmä časové závislosti čo najpresnejšie, komponenty pre meranie pokrytia boli napojené priamo na monitory príslušných rozhraní implementujúcich vrstvenie. Tieto monitory posielajú položku na analýzu v rovnakom takte, ako boli prijaté na rozhraní. Ak sa rozhraním zapíše niekoľko položiek za takt, na analýzu sa posielajú v poradí od položky na najnižšej pozícií. Diagram 8.1 znázorňuje napojenie vytvorených komponent pokrytia do prostredia *Memory Connection*.



Obr. 8.1: Komponenty funkčného pokrytia komunikácie s pamäťou.

Komponenta *Memory Operations Coverage* číta aktuálne prijaté hlavičky požiadavkov a zapísané hlavičky odpovedí. Pokrytie zbiera pre počet požiadavkov čakajúcich na odbavenie. Zaznamenáva parametre požiadavkov aj odpovedí. Prichádzajúce požiadavky na čítanie si uloží do asociatívneho poľa, kľúčom sú hodnoty *Tag* a *UnitID*. Z poľa ich vyberie po zapísaní poslednej časti odpovede. Po uložení si pre každý požiadavok postupne zaznamenáva počet odpovedí, ktoré sa odoslali na daný požiadavok. Minimálny počet je 1, maximálny je daný maximálnou veľkosťou požiadavku a parametrom *RCB*. Platí, že veľkosť prvej a poslednej odpovede môže byť nezarovnaná vzhľadom na *RCB*. Ostatné odpovede môžu mať veľkosť násobku *RCB*, preto sa maximálny počet odpovedí určí výrazom $MRRS/RCB+1$. Maximálny počet odpovedí možno dosiahnuť len u požiadavku s veľkosťou blízkou maximálnej veľkosti, ktorý nie je zarovnaný na *RCB*. Počet častí odpovede sa generuje pomocou uniformného rozloženia. Maximálny počet ostal väčšinou nepokrytý. Vyššie pokrytie možno

dosiahnuť upravením váh pri generovaní počtu častí. Preto som prostredie upravil, aby sa počet generoval v rozličných režimoch. Všetkým režimom som priradil rovnakú váhu, okrem náhodného, tomu som priradil vyššiu. Na snímku 8.2 možno vidieť počet odpovedí po úprave.

Obr. 8.2: Snímok pokrytia počtu odpovedí na požiadavok.

S uloženou položkou do poľa sa uloží aj čas prijatia, pomocou ktorého možno určiť počet taktov, za ktorý sa vrátili všetky časti odpovede. Zaznamenáva sa tiež množstvo požiadavkov na čítanie, ktoré v danom čase čakajú na odbavenie. Ich množstvo možno určiť z aktuálneho počtu položiek v asociatívnom poli. Snímok 8.3 zobrazuje ich počet pri prijatí nového požiadavku, pokrytie sa zbiera oddelene pre požiadavky na čítanie a zápis. Na snímku možno vidieť, že v prípade prijatia požiadavku na čítanie čaká na odbavenie viac ako 40 ďalších požiadavkov v $93,2\%$ prípadov. U požiadavku na zápis v $99,5\%$ prípadov.

Obr. 8.3: Snímok pokrytia počtu požiadavkov čakajúcich na odbavenie.

Pri generovaní náhodného poradia požiadavku som použil niekoľko režimov, pričom každý režim mal rovnakú pravdepodobnosť použitia. V reakcii na vysoký počet požiadavkov som priradil vyššiu pravdepodobnosť režimu, kedy sa nový požiadavok pri odbavení nepredbehne, alebo sa predbehne len o niekoľko položiek. Snímok 8.4 zobrazuje počet položiek po upravení pravdepodobnosti predbehnutia. Početnosť prípadov, kedy viac ako 40 požiadavkov čaká na odbavenie sa znížila na 90% u požiadavkov na čítanie. Pri zápise nedošlo k žiadnemu zlepšeniu. Pri ladení som si všimol, že často sú v simulácii situácie, kedy čaká na odbavenie množstvo požiadavkov a kedy čaká minimum. Preto som vytvoril nový bod pokrytia zaznamenávajúci čas medzi prijatím požiadavkov, jeho pokrytie je zobrazené na snímku 8.5. Zo snímku možno vidieť, že až 87% požiadavkov prišlo do 10 taktov po predchádzajúcom. Systém *DMA Medusa* pracuje tak, že po prečítaní nových deskriptorov odošle požiadavky do pamäte. Požiadavky sa posielajú v zhlukoch po každom prečítaní deskriptorov. Tým je ovplyvnený aj počet čakajúcich požiadavkov.

Bod pokrytia na dobu vrátenia odpovede je ovplyvnený počtom požiadavkov poslaných naraz, preto väčšina požiadavkov čakala vo na odbavenie viac ako 100 taktov. Tento bod odhalil aj, že požiadavok sa nikdy neodbaví hneď v nasledujúcom takte po prijatí. Po hlbšej analýze som zistil, že to je dané nízkoúrovňovou sekvenciou *MVB*, ktorá si položky na zápis

Obr. 8.4: Snímok pokrytia počtu požiadavkov čakajúcich na odbavenie po úprave.

Obr. 8.5: Snímok pokrytia doby medzi prijatím nových pamäťových požiadavkov.

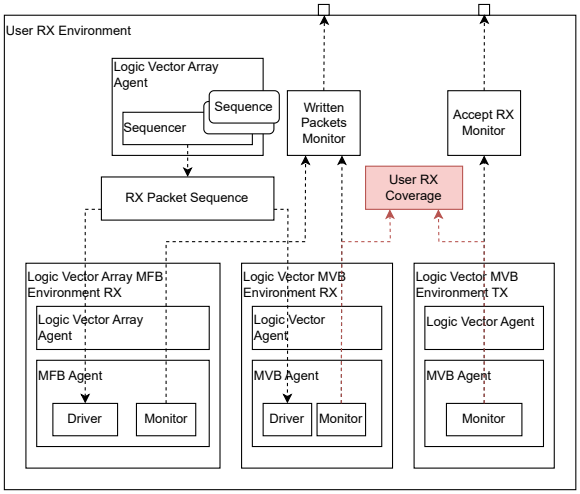
získava z vysokoúrovňovej sekvencie o takt skôr, ako ich zapíše na rozhranie. Reálna pamäť má typicky dobu odpovede niekoľko tisíc taktov a zároveň ďalšiu latenciu pridá modul *PTC*. Po zvážení týchto skutočností som sa rozhodol nevenovať oprave s vysokou prioritou.

U každého požiadavku sa na základe hlavičky zbierajú štatistiky o veľkosti a o zarovnaní. Veľkosť sa určuje aj v počte *Dwords*, aj v bajtoch. Pri dĺžke v bajtoch sa odčítajú prvé a posledné nevalidné bajty (*FirstIB*, *LastIB*). U zarovnania sa pokrývajú hodnoty *FirstIB* a *LastIB*. Pre nevalidné bajty sa zbiera aj kombinované pokrytie (*Cross coverage*), kombinujú sa aj s veľkosťou požiadavku. U odpovedí sa ukladá ich dĺžka. Dĺžka je od 1 bajtu do veľkosti *RCB*, následne potom násobky *RCB* až do veľkosti *MPS*. Hodnoty parametrov požiadavkov aj odpovedí sú pokryté dostatočne.

Komponenta *Completitions Interface Coverage* monitoruje zasielané hlavičky a rámce odpovedí. Vďaka náhodnému generovaniu medzier pri ich zápise na rozhranie má nastávať, že sa zapíše niekoľko hlavičiek bez príslušných rámcov a naopak, koľko rámcov bez hlavičiek. Zapísané hlavičky aj dátové rámce sa v rámci komponenty ukladajú do fronty. Pri každom zápise hlavičky sa zbiera informácia, koľko dátových rámcov sa pred tým odoslalo bez hlavičky, t.j. koľko rámcov sa nachádza vo fronte. Po zápise hlavičky sa z fronty odoberie 1 rámec. Ak je fronta prázdna, hlavička sa uloží do odpovedajúcej fronty. Rovnaký princíp sa používa aj u merania počtu hlavičiek zapísaných bez príslušných rámcov. Ich počty sú v oboch prípadoch pokryté dostatočne.

8.2 Prenášané pakety rozhraniami k sieťovej karte

Na rozhraniach k sieťovej karte je možné sledovať, aké pakety sa vygenerovali na zápis do pamäte a naopak, aké boli odoslané softvérom. Body pokrytia sú implementované osobitne pre smer *RX* a *TX*. Diagram 8.6 zobrazuje napojenie analytickej komponenty *User RX*



Obr. 8.6: Napojenie komponenty *User RX Coverage* na analytické porty.

Coverage. Komponenta monitoruje hlavičky a dátové rámce odoslané na rozhrania modulu *DMA*. Podobne, ako pri odosielaní odpovedí na zápisy, pokrýva sa, koľko hlavičiek bolo zapísaných bez dátových rámcov a koľko rámcov bez hlavičiek. Z hlavičiek sa zbierajú informácie o počte zaslaných paketov pre jednotlivé kanály, počet paketov s príznakom *Discard* v *log*.1, počet paketov určených na zahodenie pre jednotlivé kanály, veľkosť paketov. Zbiera sa tiež informácia o počte zapísaných paketov pre jeden kanál za sebou. Počet paketov pre jednotlivé kanály je pomerne rovnomerný, zapíše sa približne 7 000 paketov pre každý z nich. Príznak *Discard* je nastavený u 12% paketov. Počty zahodených paketov pre každý kanál sú tiež podobné medzi sebou. Snímok 8.7 zobrazuje pokrytie počtu paketov zapísaný v rade pre jeden kanál.

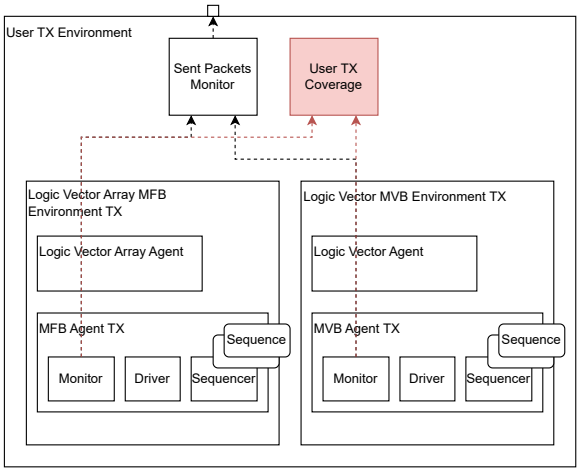
CVP cp_num_packets_for_same_channel_in_row	100.00%	100	100.00%	✓
bin one_packet	23273	1	100.00%	✓
bin two_to_nine_packets[2]	1470	1	100.00%	✓
bin two_to_nine_packets[3]	97	1	100.00%	✓
bin two_to_nine_packets[4]	13	1	100.00%	✓
bin two_to_nine_packets[5]	5	1	100.00%	✓
bin two_to_nine_packets[6]	7	1	100.00%	✓
bin two_to_nine_packets[7]	6	1	100.00%	✓
bin two_to_nine_packets[8]	8	1	100.00%	✓
bin two_to_nine_packets[9]	7	1	100.00%	✓
bin more_than_ten_packets	76	1	100.00%	✓

Obr. 8.7: Snímok pokrytia počtu paketov pre jeden kanál zapísaných za sebou.

Pri veľkosti paketov som osobitne zbieral pokrytie pre pakety s minimálnou a maximálnou veľkosťou, ďalej veľké pakety. Ich veľkosť je daná rozsahom medzi maximálnou a minimálnou veľkosťou. Za veľký paket sa považuje 20% najväčších. Rovnakým princípom sa zaznamenávali aj malé pakety. Pakety s maximálnou a minimálnou veľkosťou boli pri väčšine meraní nepokryté. Predošlá kapitola spomína, že pre generovanie paketov je použitá existujúca knižnica sekvencií. Išlo teda o chybu návrhu, kedy som nesprávne usúdil, že okrajové veľkosti budú dostatočné pokryté existujúcimi sekvenciami. Generovanie väčšieho množstva malých, prípadne veľkých paketov je pri tom vhodné na otestovanie situácií, kedy sa môže preplniť niektorá z vnútorných vyrovnávacích pamätí a podobne. V reakcii na chýbajúce pokrytie som preto vytvoril nové sekvencie, ktoré generujú minimálne, maximálne veľkosti a tiež len 10% najväčších a najmenších paketov. Sekvencie som pridal do knižnice.

Snímok 8.8 zobrazuje pokrytie po ich pridaní. V rámci knižnice som naviac znížil minimálnu dĺžku sekvencie na 1 paket. Tým sa otestujú aj prípady, kedy sa na rozhranie zapíše napríklad množstvo veľkých paketov, jeden malý a následne zase niekoľko veľkých.

Obr. 8.8: Snímok pokrytia veľkosti paketov po pridaní dodatočných sekvencií.



Obr. 8.9: Napojenie komponenty *User TX Coverage* na analytické porty.

Diagram 8.9 zobrazuje napojenie komponenty *User TX Coverage*. Komponenta monitoruje odoslané hlavičky a dátové rámce z rozhraní *User TX*. Podobne, ako pri prijímaní paketov, zbiera pokrytie pre veľkosť paketu, číslo kanála a počet odoslaných paketov za sebou z jedného kanála. Počet odoslaných paketov pre každý kanál je určený počtom transakcií pri spustení testu. Pre veľkosti paketov na odoslanie boli pridané rovnaké sekvencie, ako pre prijímanie. Pokrytie je teda podobné. Na počet odoslaných paketov za sebou má vplyv najmä náhodné spúšťanie a zastavovanie kanálov. Snímok 8.10 zobrazuje pokrytie počtu paketov pre jeden kanál za sebou. Pri odosielaní paketov sa pokrýva aj počet odoslaných hlavičiek bez príslušných dátových rámcov a naopak, počet odoslaných rámcov bez hlavičiek. Počet zapísaných rámcov bez hlavičiek je pokrytý úplne, počet hlavičiek je čiastočne nepokrytý. Po konzultácii s dizajnérom sme prišli k záveru, že modul *DMA* najskôr odosiela dáta paketov, až následne príslušné hlavičky.

8.3 Pokrytie rozhraní MVB a MFB

Prostredie obsahuje funkčné pokrytie aj pre jednotlivé signály týchto rozhraní. Body pokrytia sú implementované v komponentách *MVB IFC Coverage* a *MFB IFC Coverage*. Komponenty sú inštancované pre každé rozhranie, pripájajú sa na nízkoúrovňové monitory. Tieto monitory posielajú transakcie každý hodinový takt. U rozhrania *MVB* sa zbiera počet taktov, kedy boli porty *SRC_RDY* a *DST_RDY* v logickej 1 a 0, počet taktov, koľko zotrvali v rovnakej hodnote aj počet taktov, kedy sa *SRC_RDY* muselo držať v *log.*1, pretože

Obr. 8.10: Snímok pokrytia paketov pre jeden kanál odoslaných za sebou.

DST_RDY bolo nastavené v *log*.0. Ďalej bol pokrytý počet validných položiek v jednom slove a ich rôzne rozloženie. Rôzne rozloženie zodpovedá rozličným hodnotám portu *VLD*, pre každú hodnotu sa zbieralo pokrytie osobitne. U rozhrania *MFB* sa pokrývajú rovnaké situácie pre signály *SRC_RDY* a *DST_RDY*, ako v prípade *MVB*. Naviac sa zbiera pokrytie počtu začiatkov a koncov rámcov v rámci slova. Pre každý región sa následne pokryjú všetky pozície začiatku a konca rámca (na základe portov *SOF_POS* a *EOF_POS*).

Rozhrania *MVB Up* a *User TX MVB* môžu za takt preniesť dve položky. Pokrývajú sa len situácie, kedy je validná len prvá položka slova, alebo obidve položky. Nikdy sa u týchto rozhraní nepokryje situácia, kedy je prvá položka nevalidná a druhá validná. Dané to je spôsobom, akým modul *DMA* skladá hlavičky do slov. Ostatné body boli pokryté dostatočne až na rozhranie *User RX*, kde boli porty *DST_RDY* počas celej simulácie v *log*.1. Pokrytie sa zbieralo na hardvérovej konfigurácii so zakázaným blokujúcim režimom. Preto som dodatočne zmeral pokrytie na konfigurácii s povoleným blokovaním. Ak v tomto režime dôjde pri prijímaní paketov k preplneniu vnútorných vyrovnávacích pamätí, signály *DST_RDY* sa nastavia do *log*.0. Pre meranie som zvolil konfiguráciu s jedným koncovým bodom a ôsmimi kanálmi pre každý smer. Signál *DST_RDY* na rozhraní *MFB* bol nastavený v *log*.0 približne 30% taktov. Na rozhraní *MVB* však ostal celý čas nastavený v *log*.1. Po konzultácií s dizajnérom sme prišli k záveru, že tento signál sa môže nastaviť do *log*.0 v prípade, kedy sa zapíše niekoľko hlavičiek paketov bez príslušných dátových rámcov. Ako bolo spomenuté v kapitole 7, jednotlivé hlavičky a dátové rámce sa na rozhrania zapisujú s náhodnými medzerami prostrediami implementujúcimi vrstvenie. Pakety sú generované komponentou *User RX Packet Generator*. Táto komponenta predáva hlavičky a dátové rámce prostredníctvom schránok. Veľkosť schránky určuje teorerický limit maximálneho počtu hlavičiek a rámcov, ktoré sa môžu navzájom predbehnúť. Komponenta *User RX Packet Generator* generuje nové pakety vždy až kým sa aspoň jedna zo schránok nezaplní. Ak by veľkosť schránky bola neobmedzená, simulácia sa zasekne v jednom čase. Pri veľkosti 1 by sa hlavičky s rámcami zapisovali s minimálnym rozdielom v čase. Po zvýšení veľkosti na 1 000 položiek sa port *DST_RDY* začal nastavovať do *log*.0.

8.4 Zhodnotenie implementovaného pokrytia

V tejto kapitole bolo popísané vytvorenie a analýza funkčného pokrytia. U každej implementovanej komponenty pre pokrytie bol naznačený základný princíp zbierania pokrytia a napojenie na monitorované rozhrania. Pokrytie je zamerané na rozhrania pamäte a rozhrania k sieťovej karte. U všetkých bodov bolo uvedené namerané pokrytie. Ak bol niektorý bod pokrytý nedostatočne, kapitola uviedla zdôvodnenie. Následne bola nepokrytým bodom priradená priorita a u bodov s vysokou prioritou bol upravený stimul. Vďaka spätnej

väzbe od funkčného pokrytia boli pridané chýbajúce sekvencie, ktoré generujú pakety s minimálnou a maximálnou veľkosťou. Ďalej bolo zistené nedostatočné pokrytie pri počte odpovedí na pamäťový požiadavok. Po zistení nedostatku som upravil generovanie tak, aby sa častejšie generoval maximálny počet odpovedí. Na záver bola upravená veľkosť schránok použitých pri zapisovaní paketov na prijatie.

U pamäťového modelu bol detekovaný vysoký počet požiadavkov čakajúcich na odbavenie. V reakcií na to som znížil pravdepodobnosť predbiehania požiadavkov pred odbavením. Počet čakajúcich požiadavkov klesol, no nie výrazne. Po hlbšej analýze som určil hypotézu, že systém *DMA* posiela pamäťové požiadavky v zhlukoch, typicky po prečítaní nových deskriptorov. Na základe hypotézy som pridal ďalší bod pokrytia, ktorý určuje počet taktov medzi príchodom jednotlivých požiadavkov. Ukázalo sa, že 87% požiadavkov príde do 10 taktov po predchádzajúcom a 65% požiadavkov v rovnakom alebo nasledujúcom takte. To má vplyv na vysoký počet čakajúcich požiadavkov.

Vytvorené pokrytie možno ďalej rozšíriť na komunikáciu so softvérovými ovládačmi. Mohlo by sa zbierať buď priamo v sekvenciach softvérových ovládačov alebo sledovaním komunikácie na zbernici *MI*. Pokrytie by bolo možné implementovať aj nad subkomponentami verifikovaného systému vrátane komunikácie medzi nimi. Vďaka tomu by sa mohol definovať stimul, ktorý by kvalitnejšie otestoval správanie systému v okrajových situáciach ako je napríklad preplnenie niektorej vnútornej vyrovnávacej pamäte.

Kapitola 9

Záver

V práci som vytvoril verifikačné prostredie pre systém *DMA Medusa* podľa metodiky *UVM*. Úlohou prostredia je overiť, že systém funguje správne podľa jeho špecifikácie. Prostredie vychádza z predošlého, ktoré bolo vytvorené podľa internej metodiky združenia *CESNET*. V novom prostredí sa podarilo implementovať všetky mechanizmy pôvodného, naviac sa podarilo rozšíriť pamäťový model systému. Po implementácií prostredia bolo vytvorené aj funkčné pokrytie, vďaka ktorému bolo možné získať prehľad o generovanom stimule a jeho nedostatky vylepšiť. Pri vytvorení prostredia sa odhalili dve funkčné chyby na strane hardvéru.

Na začiatku práce som sa zameral na štúdium funkčnej verifikácie a metodiky *UVM*. Následne som prešiel k štúdiu konceptu multi zberníc. Multi zbernice sú v systéme *DMA Medusa* použité pre komunikáciu medzi vnútornými komponentami systému aj na jeho vonkajších rozhraniach. Pri štúdiu som sa zameral ako na funkčné vlastnosti, tak aj na ich rýchlostné parametre. Potom som prešiel k naštudovaniu informácií o systéme *DMA*. Najskôr som sa zameral na rozličné koncepty prenosu paketových dát medzi softvérom a hardvérom, následne som prešiel k štúdiu samotného systému. U systému som sa zameral najmä na spôsob komunikácie s okolím. Snažil som sa predovšetkým získať informácie, akým princípom pristupovať k jeho verifikácii a ako systém funguje v okrajových situáciach. Považoval som pritom za dôležité zistiť všetky potrebné informácie skôr, ako som prešiel k štúdiu predošlej verifikácie systému. Inak by hrozilo, že by som replikoval prípadné koncepčné chyby predošlej verifikácie. Informácie o systéme som zbieral do interného dokumentu. V budúcnosti by sa informácie z tohto dokumentu mali zakomponovať do špecifikácie systému. Po naštudovaní základných princípov jeho fungovania som prešiel k analýze predošlého prostredia. Išlo o prostredie, ktoré vykonáva funkčnú verifikáciu. Systém sa testoval pomocou náhodného stimulu. Pri analýze som zistil, že predošlá verifikácia vhodne implementovala princípy paketových prenosov. Za najväčší nedostatok som označil absenciu verifikačného plánu. Po analýze predošlého prostredia som naštudoval princíp použitia existujúcich verifikačných komponent vytvorených podľa metodiky *UVM*, ktoré som v novom prostredí použil.

Po štúdiu potrebných materiálov a analýze predošlého prostredia som prešiel k definovaniu požiadavkov na nové prostredie. Požiadavky som rozdelil na požadované vlastnosti generovaného stimulu a vlastnosti, ktoré má nové prostredie kontrolovať. Takto definované požiadavky svojím obsahom zodpovedajú obsahu verifikačného plánu. Požiadavky teda vychádzali z naštudovaných informácií o systéme, jeho špecifikácie a boli tiež inšpirované princípmi použitými v predošlom prostredí. Jedným z požiadavkov, ktoré som identifikoval naviac oproti predošlej verifikácií, bola potreba náhodného poradia odbavenia požiadavkov do pamäte tak, ako to umožňuje špecifikácia zbernice *PCIe*.

Po definovaní požiadavkov som prešiel k implementácii systému. Na základe požiadavkov som mohol navrhnúť prostredie, ktoré ich umožnilo implementovať. Prostredie implementuje všetky mechanizmy pôvodného prostredia, prípadne implementuje alternatívne, ktoré sú k pôvodným ekvivalentné. Prostredie bolo rozšírené o vyššie spomenuté náhodné poradie odbavenia požiadavkov. Vďaka nemu je možné odhaliť prípadné chyby, ktoré by mohli nastať po napojení systému *DMA Medusa* na pamäťový systém tretej strany. Po implementácií som pristúpil k overeniu generovaného stimulu pomocou funkčného pokrytia. Pri funkčnom pokrytí som sa zameral na rozhrania pre komunikáciu s pamäťou a rozhrania smerom k sieťovej karte. Odhaleným nedostatkom som priradil rozličnú prioritu a nedostatky s najvyššou som sa pokúsil odstrániť. Navrhol som aj ďalšie rozšírenia implementácie a funkčného pokrytia.

Počas vytvárania prostredia boli odhalené dve funkčné chyby vo verifikovanom systéme. Prvá chyba sa objavila po vytvorení pamäťového modelu, ktorý podporuje predbiehanie odbavenia požiadavkov na čítanie zápisovými, pričom požiadavok na čítanie sa môže vykonať po niekoľkých častiach a len niektoré z častí budú predbehnuté zápisom. V druhom prípade išlo o situáciu, kedy modul *DMA* používa pre pamäťové požiadavky obmedzený počet hodnôt *Tag*. Kvôli pamäťovému modelu došlo k situácií, kedy sa všetky hodnoty použili a čakalo sa na odbavenie jednotlivých požiadavkov. Táto situácia nebola zohľadnená v kontrolnej logike jednej z nadväzujúcich komponent.

Literatúra

[1] *Semiconductor Engineering: Knowledge Center - Formal Verification* [online]. Semiconductor Engineering [cit. 2022-12-28]. Dostupné z: https://semiengineering.com/knowledge_centers/eda-design/verification/formal-verification/.

[2] *Semiconductor Engineering: Knowledge Center - Verification* [online]. Semiconductor Engineering [cit. 2022-12-28]. Dostupné z: https://semiengineering.com/knowledge_centers/eda-design/verification/.

[3] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*. 2020, s. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[4] Arditi, L., Sargent, P., Aird, T. a Choquin, L. A formal-based approach for efficient RISC-V processor verification. *Verification horizons*. 2023, zv. 19, s. 4. Dostupné z: https://verificationacademy.com/verification-horizons/march-2023-volume-19-issue-1/formal-based-approach-for-efficient-risc-v-processor-verification.

[5] Allan, G., Chidolue, G., Ellis, T., Foster, H., Horn, M. et al. *Coverage Cookbook* [online]. Mentor, 2019 [cit. 2023-5-5]. Dostupné z: https://verificationacademy.com/cookbook/coverage.

[6] CESNET. *Verifikačné prostredie systému DMA Medusa* [online]. 2018 [cit. 2023-4-15]. Dostupné z: https://gitlab.liberouter.org/ndk/ndk-mod-dma-medusa/.

[7] CESNET. *Řízení řadiče a komunikace se softwarem* [online]. 2020 [cit. 2023-4-15]. Dostupné z: https://redmine.liberouter.org/projects/tmc/wiki/Dma_medusa_sw.

[8] Kekely, L., Cabal, J., Puš, V. a Kořenek, J. Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, s. 49–56. DOI: 10.1109/DSD51259.2020.00020.

[9] Kubálek, J. *Vysokorychlostní paketové DMA přenosy do FPGA* [online]. Brno, Vysoké učení technické v Brně, Fakulta informačních technologií, 2020. [cit. 2022-12-27]. Diplomová práce. Vedúci práce Martínek, T. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/129311.

[10] CESNET. *MI bus specification* [online]. 2021 [cit. 2023-1-1]. Dostupné z: https://cesnet.github.io/ofm/comp/mi_tools/readme.html.

[11] Siemens. *Universal Verification Methodology UVM Cookbook* [online]. 2018 [cit. 2023-1-9]. Dostupné z: https://verificationacademy.com/cookbook/uvm.

[12] *Universal Verification Methodology (UVM) 1.2 User's Guide* [online]. UVM Working Group and Accellera Systems Initiative, 2015 [cit. 2023-1-7]. Dostupné z: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[13] Foster, H. *2022 Wilson Research Group Functional Verification Study* [online]. 2022 [cit. 2023-1-12]. Dostupné z: https://verificationacademy.com/seminars/2022-functional-verification-study.