VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

VERIFIKACE FUNKČNÍCH BLOKŮ PRO FPGA

VERIFICATION OF FUNCTION BLOCKS FOR FPGA

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. Daniel Kříž

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. Petr Jedlička

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Daniel Kříž ***ID:*** 203269

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

NÁZEV TÉMATU:

Verifikace funkčních bloků pro FPGA

POKYNY PRO VYPRACOVÁNÍ:

Podrobně prostudujte moderní metodiky (například UVM a CocoTB) pro verifikaci integrovaných obvodů a navrhněte verifikační prostředí pro digitální obvod (dle potřeby na úrovni jednotky nebo na úrovni menších celků), který řeší příjem a vysílání ethernetových paketů v FPGA. Navržené verifikační prostředí implementujte pomocí vhodně zvolené verifikační metodiky a jeho funkčnost ověřte základní verifikací digitálního obvodu. Následně proveďte důkladnou verifikaci zvoleného obvodu pro příjem a vysílání ethernetových paketů. V závěru práce uveďte výhody a nevýhody zvoleného řešení a diskutujte možná rozšíření. Při návrhu i realizaci uvažujte pokrytí testovaného kódu. Pro realizaci lze použít sadu skriptů a již existujících komponent v databázi společnosti CESNET z.s.p.o.

DOPORUČENÁ LITERATURA:

[1] SPEAR, Chris, TUMBUSH, Greg. SystemVerilog for Verification, Springer US, 2012. ISBN 978-1-4614-0714-0.

[2] SUTHERLAND, Stuart. RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design, CreateSpace Independent Publishing Platform, 2017. ISBN 978-1546776345.

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

Vedoucí práce: Ing. Petr Jedlička

Konzultant: Ing. Radek Iša, CESNET z.s.p.o

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Tato diplomová práce je věnována problematice verifikací funkčních bloků pro FPGA. V teoretické části práce je popsán koncept verifikace, verifikačních metodologií, které poskytují potřebné nástroje pro otestování daného návrhu, a na závěr je diskutovaná problematika Ethernetu a jeho odlišnosti oproti nízkolatenční variantě. Cílem praktické části diplomové práce je na základě získaných teoretických znalostí a vybrané verifikační metodologie sestrojit verifikační prostředí, provést důkladnou verifikaci nízkolatenční fyzické vrstvy Ethernetu a na závěr realizovat měření latence a propustnosti tohoto obvodu.

KLÍČOVÁ SLOVA

Ethernet, Verifikace, UVM, SystemVerilog, SVA

ABSTRACT

This master thesis is devoted to the issue of verification of function blocks for FPGA. The teoritical part of thesis describes the concept of verification, verification methodologies that provide the necessary tools for testing the design, and finally discusses the issue of Ethernet and its differences from the low-latency variant. The aim of the practical part of the master thesis is based on the acquired theoretical knowledge and selected verification methodology to build a verification environment, perform a thorough verification of the low-latency physical layer of Ethernet and finally measure the latency and throughput of this circuit.

KEYWORDS

Ethernet, Verification, UVM, SystemVerilog, SVA

KŘÍŽ, Daniel. *Verifikace funkčních bloků pro FPGA*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 112 s. Diplomová práce. Vedoucí práce: Ing. Petr Jedlička

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Daniel Kříž
VUT ID autora:	203269
Typ práce:	Diplomová práce
Akademický rok:	2021/22
Téma závěrečné práce:	Verifikace funkčních bloků pro FPGA

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval technickému vedoucímu panu Ing. Radkovi Išovi a vedoucímu diplomové práce panu Ing. Petru Jedličkovi, za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V síťové infrastruktuře je dnes kladen důraz nejen na vysoké přenosové rychlosti, ale i na nízký reakční čas, jinak též latenci těchto systémů. Tato skutečnost stojí za vznikem tzv. nízkolatenčních obvodů používajících se například k obchodování na burze, kde je potřeba v případě výhodného nákupu rychle reagovat. Vhodným řešením, jak takovýchto výstupů docílit je použití programovatelných vysokorychlostních čipů FPGA (Field Programmable Array). Jedna z částí tohoto čipu, která výrazně ovlivňuje latenci je Ethernet. Aby bylo možné docílit požadovaných výsledků, je zapotřebí provést změny v logice tohoto systému, čímž dojde i ke změně chování této technologie. Abychom ověřili, zda-li po úpravách byla zachována spolehlivost upravované technologie, je nutné provést důkladné testování.

V dnešní době existuje velké množství ověřovacích technik, které lze použít pro ověření funkčnosti těchto obvodů, patří mezi ně například simulace a testování, Funkční verifikace a Formální verifikace. Ačkoliv simulace a testování vypadá staromódně, má stále velké využití zejména při ladění v prvotní fázi implementace. Častější a důkladnější forma ověřování je pak verifikace, která slouží k podrobnějšímu prověření chování daného systému pomocí náhodně generovaných stimulů. Tato práce se dále podrobněji zaměřuje právě na Funkční verifikace.

Cílem této práce je vytvoření verifikačního prostředí na základě zvolené verifikační metodologie a za jeho pomoci provést důkladnou verifikaci nízkolatenčního Ethernetového PHY.

První část je zaměřená na teoretickou část, kterou je nutno prostudovat před realizací praktické části. V první kapitole jsou obecně popsány verifikační techniky a některé jejich formy, zejména Formální a Funkční verifikace. Následně je důkladně vysvětleno, jak verifikace vzniká a jaké náležitosti musí splňovat. Na závěr kapitoly jsou vysvětleny základy programovacího jazyka System Verilog. Další část je věnována verifikačním metodikám, zejména jejich porovnání a podrobnému popisu vybrané metodologie UVM. Poslední část teoretického úvodu je věnována studiu problematiky Ethernetu, zejména funkčních bloků MAC a PCS a jejich podrobnému popisu.

Plynule na předchozí teorii navazuje praktická část, která je zaměřená na realizaci verifikačního prostředí, jež je následně využito pro verifikaci nízkolatenční fyzické vrstvy Ethernetu (ETH PHY). První fáze je věnována studiu funkcionality verifikovaného obvodu, zejména jeho blokovému schématu a funkcionálním změnám oproti běžnému Ethernetu. Další část je věnovaná návrhu verifikačního prostředí pro rozhraní LII a PMA. Následující kapitoly jsou pak věnovány realizaci verifikací jednotlivých podkomponent ETH PHY a následně i samotného celku. Pro každou komponentu je vysvětleno jakým způsobem je verifikována, posléze jsou debatovány

zjištěné výsledky a odhalené chyby. V případě verifikace celého ETH PHY jsou zde navíc shrnuty výsledky, které byly získány měřením propustnosti a latence. Na závěr jsou shrnuty a vyhodnoceny výsledky diplomové práce a je zde uvedeno jakým směrem by se dále práce mohla ubírat do budoucna.

1 Verifikační techniky

Tato kapitola se zabývá nastíněním problematiky nejznámějších typů verifikačních technik. První bude popsána Formální verifikace, kde bude vysvětlen její princip a nejčastěji používané metody pro její realizaci. Dalším odvětvím, které bude popsáno je Funkční verifikace, kde bude nastíněn zejména proces vytváření takovéto verifikace. Na závěr bude popsán programovací jazyk, který je hojně využíván k vytváření verifikací.

1.1 Formální verifikace

Formální verifikace je soubor technik, které využívají statickou analýzu založenou na matematických modelech. Na rozdíl od ostatních typů testování se formální verifikace liší tím, že využívá formální metody k ověření korektního chování obvodu. Při úspěšném verifikování určitého chování si tedy následně můžeme být jisti, že verifikovaný obvod danou funkcionalitu splňuje. Formální verifikace, která ověřuje korektní chování obvodu na všech vstupech, je definovaná jako NP úplný problém, s počtem vstupů tedy roste čas na vyřešení problému exponenciálně. Aplikace této formy verifikace je tudíž na větší obvody časově náročná. [1]

1.1.1 Nejznámější techniky formální verifikace

Existuje velké množství metod formální verifikace, mezi nejznámější patří Model checking, Theorem Proving a Static Analysis, které jsou popsány níže.

Model Checking

Formální verifikační metoda, která porovnává abstraktní reprezentaci systému (modelu) se specifikací reálného systému. Model je tvořen stavovým automatem, jenž obsahuje stavy definované verifikačním technikem. Postupně se stavy prochází a zjišťuje se zda, model odpovídá zadaným požadavkům. Pokud požadavky nejsou splněny, model je upraven (přidají se stavy, které nebyly ošetřeny). Upravuje se do té doby, dokud nejsou pokryty všechny možné stavy, to v praxi znamená, že neexistuje stav, kdyby byly požadavky na systém porušeny. Často se používá k odhalení dobře skrytých chyb. [1]

Theorem Proving

Deduktivní metoda, která je velice podobná matematické metodě dokazování vět, počínaje axiomy a odvozováním dalších skutečností pomocí pravidel odvozování.

Tento přístup je velice obecný a poloautomatický a často vyžaduje značné úsilí uživatelů. [2]

Static Analysis

Metoda s vysokou úrovní automatizace, která analyzuje zdrojový kód systému, aniž by byl spuštěn, a zároveň z něj sbírá informace o systému. Existují různé formy této analýzy (type analysis, bug pattern searching, equation dataflow analysis, constrained-based analysis nebo abstract interpretation). Nepoužívá se jen pro zjištění chyb systému, ale i pro optimalizaci kódu. [3]

1.2 Funkční verifikace

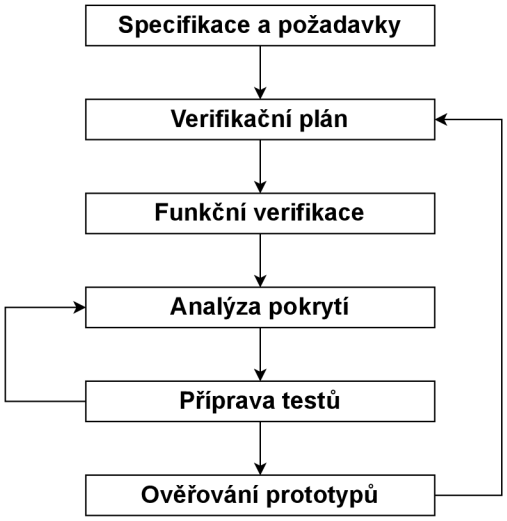
Funkční verifikace je proces, kterým se ověřuje správné chování systému dle specifikací. Tento proces je založen na simulaci, která využívá důkladnější testovací scénáře. K testování využívá přídavné funkce jako například náhodné generování vstupů (constrained-random stimulus), se kterou je souběžně využíváno pokrytí nebo také **coverage**, jenž říká, zda-li byl počet vygenerovaných stimulů dostatečný, aby ověřil chování obvodu ve většině případů. Proč je vlastně funkční verifikace potřebná? Tento typ verifikace vznikl, protože samotná simulace nebyla dost důkladná, nebylo možné pokrýt veškeré stavy systému. Formální verifikace je sice důkladná, ale příliš náročná. Funkční verifikace díky náhodnému generování různých scénářů dokáže důkladně ověřit funkčnost systému a zároveň nezabere tolik reálného času. Princip bude popsán v sekci 1.2.4. Pro implementaci funkčních verifikací se nejčastěji používá programovací jazyk System Verilog, který bude popsán ve sekci 1.2.5. [4]

1.2.1 Proces verifikace

Verifikace je složitý proces, jenž má za úkol ověřit korektní chování vyvíjeného systému (DUT - Design Under Test) tak, aby splňoval všechny požadavky, které jsou definované v jeho specifikaci. Jedná se o proces, jenž jde ruku v ruce s návrhem obvodu. V následujících podkapitolách budou vysvětleny hlavní kroky při vytváření verifikace, které jsou znázorněny na obrázku 1.1.

1.2.2 Studium specifikace a shromažďování požadavků na verifikaci

Důležitým krokem při vytváření funkční verifikace je zjištění požadavků na verifikaci a prostudování specifikace. Požadavky slouží jako základ, ze kterého se bude



Obr. 1.1: Proces verifikace [4]

odvíjet plánování verifikačního procesu, a měli by být definovány co nejdříve. Doporučuje se, aby požadavky byly ověřeny členy celého týmu, tedy vývojáři hardwaru, softwaru a verifikačními techniky. Mezi důležité požadavky patří například: popis vstupů a výstupů na základě protokolů k rozhraním, popis chování hardwarové a softwarové části, popis nesprávného chování návrhu (aby bylo možné ošetřit kontrolu funkčních chyb návrhu), seznam potenciálních krajních případů, možné chyby, ze kterých se návrh dokáže vzpamatovat (například nízká chybovost a restart návrhu), a jak na takové chyby má reagovat. Dále je důležité určit možné konfigurace návrhu. Těmi jsou například šířka dat, šířka metadat a seznam parametrů, které mohou být použity. [4]

1.2.3 Verifikační plán

Verifikační plán slouží k vytvoření přehledu o tom, co je zapotřebí otestovat. Tímto plánem je poté řízena celá verifikace. Plán neobsahuje testy, které budou použity, ale spíše uvádí, jak k problematice přistupovat. Jsou stanoveny cíle, jež je nutné splnit. Dále určuje, jakým způsobem se budou cíle řešit, jestli pomocí formální technik, simulačních, či pomocí funkční verifikace. V plánu jsou také shromažďovány a sledovány údaje o pokrytí. Verifikační plán může být ve formě tabulky, textového dokumentu atd. Skládá se z následujících částí: [5]

- **Přehled** – v této části je obsažen popis projektu a jednotlivých bloků hardwaru a softwaru. Například je zde řečeno jaká specifikace bude použita, v jakém jazyce bude verifikace napsána a co za metodologie bude použito.
- **Seznam funkcí** – obsahuje všechny funkce, které by verifikace měla ověřit.

Každý záznam by měl obsahovat unikátní jméno, popis, očekávaný výsledek, odkaz na funkci ve specifikaci a prioritu.

- **Zdroje, rozpočet a rozvrh** – v této části se nacházejí informace o časové náročnosti jednotlivých částí verifikace v člověkohodinách. Dalšími důležitými částmi je rozvrh práce pro každou fázi verifikace a informace o použitých nástrojích.
- **Verifikační prostředí** – zde se nachází detailní popis architektury Test-Benche, použitých technik, znovupoužitelných verifikačních komponent a návod, jak používat tyto komponenty.
- **Verifikační struktura** – obsahuje podrobnosti o jednotlivých úrovních a fázích verifikace.
- **Plán generování stimulů** – v tomto plánu jsou popsány použité transakce, sekvence a testovací scénáře. Každá položka v tomto plánu obsahuje unikátní identifikátor, popis stimulu, konfigurace a informace o omezeních pro dané stimuly.
- **Kontrolní plán** – popisuje očekávaný výsledek verifikace, nejčastěji zde bývá popis funkce monitoru nebo kontrolní logiky. Každá položka obsahuje unikátní jméno, identifikátor funkce a popis.
- **Plán pokrytí** – popisuje funkcionální pokrytí funkcí. Je sestaven, aby poukázal, jakým způsobem mají být implementovány jednotlivé body pokrytí ve verifikačním prostředí. Mezi důležitá pole v tomto plánu patří: skupina bodů pokrytí, popis pokrytí, název pokrytí, unikátní identifikátor, bod pokrytí a cíl pokrytí.
- **Detaily o znovupoužitelnosti komponent** – obsahuje informace o znovupoužitelných komponentách a jejich použití.

1.2.4 Verifikace systému

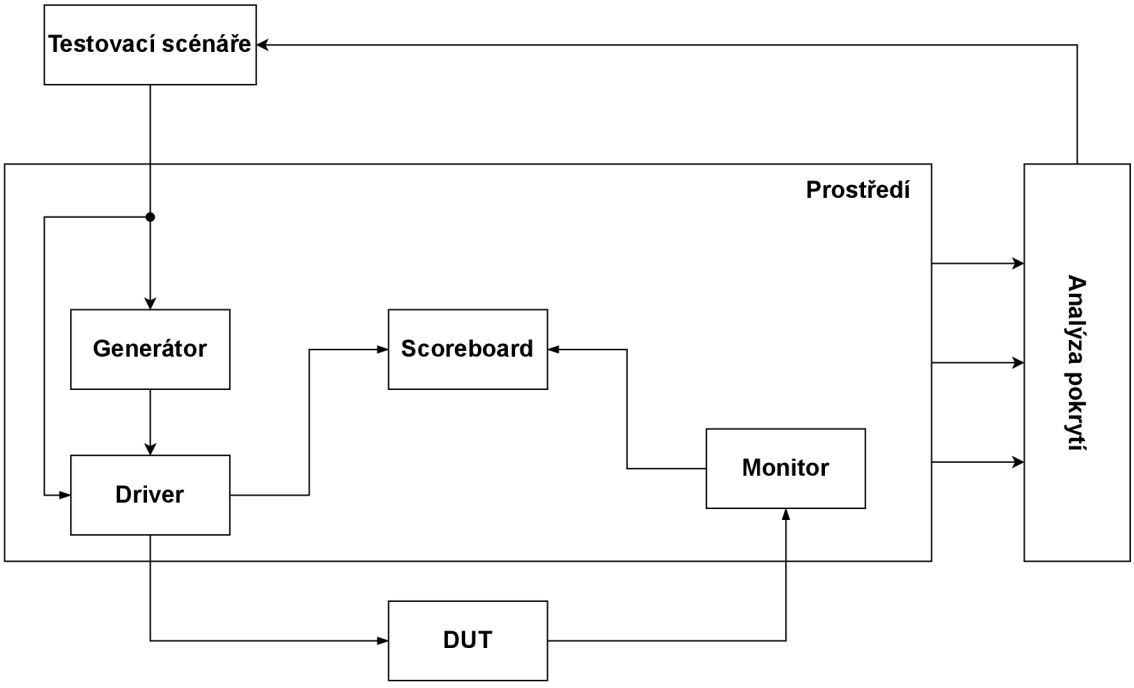
V této sekci bude popsán princip funkční verifikace a tvorba testovacích scénářů. Posléze bude popsáno verifikační prostředí a s ním spojená analýza pokrytí. [4]

Princip

Funkční verifikace je tvořena prostředím, které je rozděleno do tří hlavních vrstev: vrstva scénářů, verifikační vrstva a signálová vrstva. V nejvyšší vrstvě se nachází testovací scénáře pro danou verifikaci. Tyto scénáře definují, jakým způsobem budou generovány transakce generátorem. Počet scénářů je postupně rozšiřován na základě analýzy pokrytí, dokud nejsou pokryty všechny stavy systému. Prostřední vrstva se stará o samotnou verifikaci. Tato vrstva může obsahovat velké množství

verifikačních komponent, mezi základní patří generátor, driver, scoreboard a monitor. Generátorem jsou na základě pravidel od vyšší vrstvy generovány stimuly. Stimuly jsou data aplikovaná na DUT, mohou být reprezentovány různými způsoby, nejčastěji se ale používají tzv. transakce, uvnitř nichž je definována struktura dané datové jednotky (rámce, paketu, fragmentu atd.), ta může obsahovat například začátek paketu, konec paketu, data, meta data a další. Tato transakce je následně odeslána driveru, který transakci rozdělí na jednotlivé signály rozhraní a provede zapsání na rozhraní DUT a zároveň i do scoreboardu. V DUT se provede kód, jehož výsledkem je výstupní signálový tok. Ten je odeslán do monitoru, kde je signálový tok navzorkován a je z něj vytvořena výstupní transakce. Následně je vytvořena tzv. referenční transakce (výstupní transakce vycházející ze specifikace). Ta vznikne ze vstupní transakce, která je modelem upravena na základě specifikace. Obě tyto transakce jsou odeslány do scoreboardu, kde proběhne jejich porovnání. Z výsledku porovnaní je vyvozen výsledek verifikace (Úspěch/Neúspěch). Nejspodnější vrstva obsahuje samotný verifikovaný systém, jenž je propojen s verifikačním prostředím skrze rozhraní (sada signálů). [4]

Verifikační prostředí

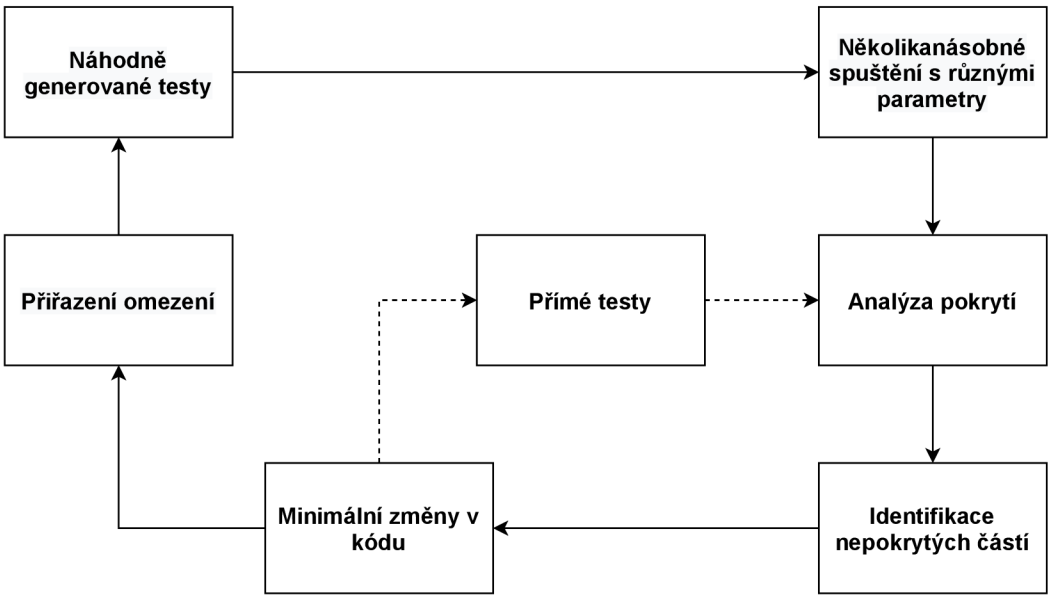


Obr. 1.2: Verifikační prostředí [4]

Verifikační prostředí, které je zobrazené na obrázku 1.2, je abstraktní objekt, ve kterém se nachází jednotlivé verifikační komponenty, které jsou zapotřebí k řešení

problému. Níže si popíšeme funkcionalitu jednotlivých částí: [6]

- **Testovací scénáře** – obsahují omezení pro generátor (parametry, podmínky atd.)
- **Generátor** – na základě testovacího scénáře generuje transakce, které jsou odesílány driveru.
- **Driver** – zapisuje transakce, které obdržel, na vstup verifikovaného systému a do scoreboardu.
- **Scoreboard** – slouží k porovnání referenčního výstupu s výstupem verifikovaného systému. Součástí scoreboardu je model, který má za úkol upravit vstupní transakci tak, aby odpovídala specifikaci. Výsledkem této operace je referenční transakce.
- **Monitor** – je komponenta, která má za úkol snímat signály na verifikovaném rozhraní a převádět je na transakce, které je následně možné používat ve verifikaci.
- **DUT** – označení pro testovaný systém.
- **Analýza pokrytí** – sbírá a analyzuje data z provedených verifikačních běhů, na základě nichž je vygenerována zpráva pokrytí. Na základě této zprávy je zjištěno, které části obvodu nebyly verifikovány. Jako reakce na tuto zprávu jsou vytvořeny další verifikační scénáře, které otestují tyto části. Princip analýzy pokrytí lze shrnout diagramem na obrázku 1.3.



Obr. 1.3: Proces analýzy pokrytí [4]

1.2.5 System Verilog

System Verilog je programovací jazyk, vycházející z jazyku Verilog. Využívá se primárně na vytváření verifikací hardwarových návrhů, ale lze ho použít i k jejich vytváření. Nabízí API pro pokrytí a ošetřování vstupů (Assertion) a DPI (Direct Programming Interface), tedy rozhraní, které lze použít k propojení jazyku SystemVerilog s cizími jazyky. Obsahuje různé techniky, které umožní verifikačnímu technikovi vytvářet verifikace snadněji. Tyto techniky budou popsány podrobněji níže [15]

- **Náhodné generování vstupů** – umožňuje uživatelům automaticky generovat vstupní stimuly pro funkční verifikaci. Tento způsob je efektivnější než přímé testování. K tomuto testování se využívají tzv. podmínky, boolovský výraz popisující vlastnosti vstupní proměnné. Tyto podmínky se využívají k nastavení náhodného generátoru, určuje se jimi omezení, které má při generování hodnot (například, že musí generovat hodnoty z rozsahu A-B, kde A a B jsou meze). V mezích jsou hodnoty vybírány náhodně. Specifikováním těchto podmínek se snaží návrhář omezit generování vstupních vektorů tak, aby pokryly většinu testovacích případů v co nejkratším čase. [8]
- **Verifikace založená na tvrzení** (Assertion-Based Verification) – technika, která umožňuje zvětšit efektivitu verifikačního prostředí. Používá se k zachycení specifického chování hardwarového návrhu. Následně prostřednictvím simulace, formální verifikace nebo emulací těchto tvrzení ověřují, zda je návrh správně implementovaný. Mezi výhody této techniky patří redukce verifikačního času, dřívější zachycení chyb a určení zdrojů chyb (Assertions jsou připnutá k prvkům návrhu, tudíž je ve většině případů jisté, kde se stala chyba). Existují tři možnosti jak tuto metodu implementovat: [9]
  - Technik, který navrhoval návrh, je vloží do zdrojového kódu.
  - Verifikační technik je vloží k souborům, které jsou součástí verifikace. V tomto případě se nejčastěji používá jazyk System Verilog Assertions (SVA).
  - Pomocí třetí strany, kdy poskytovatel standardu nebo nástroje poskytne knihovnu běžných tvrzení pro běžné případy použití. Nejčastěji používanou knihovnou je Open Verification Library (OVL), která byla vyvinuta společností Accellera.
- **Funkční pokrytí** – slouží jako metrika pro zjištění efektivnosti použitých testů. Jedná se o uživatelem nadefinovanou metriku, kterou určuje, jak moc má být návrh testován. Uživatel nadefinuje stavy systému, do kterých chce, aby se systém během testování dostal a verifikace pak sleduje jestli tato situace nastala. Pokrytí je možné definovat na 4 různých místech: [10]

  - poblíž náhodného generátoru,
  - na vstupním rozhraní návrhu,
  - uvnitř návrhu (vzorkováním vnitřních stavů návrhu),
  - na výstupním rozhraní návrhu.
- **Pokrytí kódu** – metrika, kterou zjišťujeme kvalitu verifikace. Tuto metriku sbírá simulátor a zjišťuje, jakým způsobem byl prováděn kód hardwarového návrhu, například jestli byli spuštěny všechny řádky kódu. Existuje několik typů pokrytí kódu: [7]
  - **Řádkové pokrytí** – kontroluje, zda byly provedeny všechny řádky kódu.
  - **Pokrytí tvrzení** – kontroluje, jestli byly provedeny všechny tvrzení v kódu.
  - **Pokrytí větve** – kontroluje, zda-li byly provedeny všechny větve v kódu.
  - **Pokrytí výrazů** – ověřuje, jestli byly provedeny všechny výrazy, které by mohly ovlivnit danou větev.
  - **Stavové pokrytí** – sleduje, zda-li byly aktivní všechny stavy stavových automatů.
  - **Pokrytí přepínačů** – prověřuje, zda došlo ke změně u všech proměnných.

2 Verifikační metodologie

K vytvoření verifikace, která úspěšně otestuje daný systém není jen programovací jazyk. Důležitým aspektem je použitá verifikační metodologie. Jak ale vybrat nejefektivnější metodologii? Efektivní metodologie musí splňovat dva nejdůležitější aspekty, jimiž jsou produktivita a kvalita. Taková metodologie by měla disponovat objekty a metodami, které lze efektivně využít k otestování zadaného systému. V následujících sekcích budou popsány 4 nejznámější verifikační metodologie. Metodologie budou mezi sebou porovnány a jedna z nich bude vybrána a v samostatné kapitole 2.1 popsána. [11]

- **Verification Methodology Manual (VMM)** – první úspěšně implementovaná sada technik pro vytváření znovupoužitelného verifikačního prostředí v System Verilogu. Tato metodologie byla vytvořena společností Synopsys a je základem pro ostatní metodologie. [13]
- **Open Verification Methodology (OVM)** – jedná se o verifikační metodologii pro System Verilog, která disponuje knihovnou objektů a procedur pro generování stimulů, sběr dat a kontrolu verifikačního procesu. Díky těmto objektům mohou uživatelé vytvářet znovupoužitelné verifikační prostředí, ve kterém komunikace mezi jednotlivými komponenty probíhá skrze TLM (Transaction-level Modeling) rozhraní. OVM bylo vyvinuto ve spolupráci společností Cadence a Mentor Graphics. Jedná se o open source metodiku, která běží pod licencí Apache 2.0. [12]
- **Universal Verification Methodology (UVM)** - open source knihovna System Verilogu umožňující vytvoření flexibilních a znovupoužitelných komponent. Umožňuje vytváření výkonných testovacích prostředí, které využívají náhodného generování podmíněných stimulů a funkčního pokrytí. Oproti OVM přináší například dokonalejší fázovací systém, který umožňuje uživateli ovlivnit chování verifikačního prostředí v různých fázích. Tato metodologie bude podrobněji popsána v sekci Metodologie UVM 2.1. [13]
- **CocoTB (COroutine based COsimulation TestBench)** – je open source nástroj umožňující psát simulace a verifikace HDL návrhů pomocí programovacího jazyka Python, což je velice produktivní jazyk poskytující velké množství knihoven a funkcionalit, které se hodí při vytváření testbenche. Sám o sobě neprovádí simulaci, ale využívá k tomu externí nástroje. Mezi nejznámější patří například Verilator. Funkcionálně je velice podobný verifikacím pomocí UVM. [14]

Na základě studia výše zmíněných metodologií byli vybráni dvě z nich, UVM a cocoTB, a bylo provedeno jejich porovnání. Nejprve byla otestována funkčnost cocoTB vytvořením testbenche, který měl odsimulovat chování jednoduché asyn-

chronní FIFO paměti. Implementace byla časově náročná, nejprve byla snaha vytvořit verifikaci, ale po dlouhém hledání informací o tom, jak psát verifikace v tomto prostředí, bylo zjištěno, že je to možné. Existuje na to knihovna `pyuvm`, která obsahuje komponenty podobné UVM. Avšak jedná se pouze o obálku nad UVM objekty, která je umožňuje používat v cocoTB. Jelikož by vytvoření zabralo velké množství času, byl vytvořen jen testbench a na základě něj byly vyvozeny výhody a nevýhody tohoto prostředí. Výhody: [14]

- Je napsaný v Pythonu, který není tolik komplexní na pochopení.
- Je kompatibilní se širokou škálou simulátorů.
- Existují pro tento model i open-source simulátory, nejznámější je Verilator.
- Je kompatibilní jak s návrhy napsanými ve VHDL, tak ve Verilogu.

Nevýhody:

- Není tolik rozšířený, tudíž ho nepoužívá moc firem, a tedy má špatné teoretické zázemí.
- Python je sice jednoduchý na pochopení, ale je taky pomalý, tudíž verifikace nepoběží moc rychle.
- Úzkým hrdlem jsou simulátory. Ačkoli jich podporuje širokou škálu, pro jejich použití jsou potřeba velké úpravy v Makefilu.
- Není kompatibilní se simulátorem od firmy Xilinx.

Druhou metodologií bylo UVM. Po důkladném hledání informací a studiu dokumentace se jevila jako kompaktnější a lépe použitelná. Oproti předchozí možnosti má důkladný teoretický základ, který dostatečně uvede uživatele do problematiky. Dále je dostupné fórum, kde je spousta příkladů a rad, takže je možné dohledat i nějaké použitelné informace. Tak jako k předchozímu prostředí byly vypsány výhody a nevýhody i k UVM. Výhody:

- Pokud vývojář správně napíše prostředí pro dané rozhraní, je možné jej používat na různé verifikace.
- Umožňuje oddělovat testy od testbenche, a proto je možné jejich opětovné použití.
- Je nezávislé na simulátoru.
- Poskytuje sekvence, které umožňují dobrou kontrolu nad generováním stimulů.
- Pro komplexnější prostředí poskytuje konfigurační soubory, které zjednodušují konfiguraci objektů s hlubší hierarchií komponent.
- Díky tzv. továrně (komponentě, která vytváří UVM objekty) je umožněna jednoduší modifikace komponent.
- Poskytuje nástroj na vygenerování kostry verifikace, čímž usnadňuje implementaci.

Nevýhody:

- Je náročnější na studium, ale to bylo i předchozí prostředí.

Po důkladném studiu byla tedy nakonec vybrána metodologie UVM, více informací je možné dohledat v sekci 2.1.

2.1 Metodologie UVM

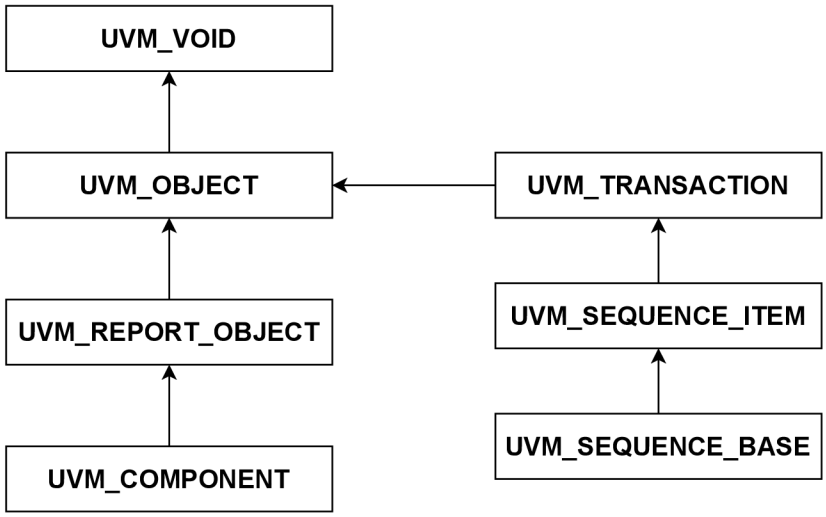
UVM (Universal Verification Methodology) je standard umožňující zrychlení vývoje a znovupoužití verifikačního prostředí v celém průmyslu. Jedná se o sadu třídních knihoven napsaných v SystemVerilogu. Vychází z OVM a má velkou podporu EDA prodejci. Přináší jistou abstrakci, kde každá komponenta ve verifikačním prostředí má svou roli. Oproti klasické verifikaci se UVM liší svou architekturou a rychlostí. Sequencer, driver a monitor jsou sjednoceny pod komponentou, které se říká agent, kde dochází ke zpracování příchozích transakcí. Generování transakcí je obdobné jako u klasické verifikace a to tak, že buď existuje generátor, nebo vysokoúrovňový agent, který generuje transakce vyšší úrovně, nejčastěji datové rámce. Nejčastěji obsahuje pole bytů. Dále je transakce odeslána pouze na jedno rozhraní, kde sídlí DUT, tam je transakce patřičně zpracována a odeslána směrem k monitoru. Do monitoru je odeslána i referenční transakce (tzn. transakce, která by měla odpovídat specifikaci), dojde k navzorkování signálů a sestrojení výstupní transakce. Obě transakce jsou následně odeslány směrem do scoreboardu, kde jsou patřičně upraveny a je vyhodnoceno, zda-li jsou obě transakce stejné. Na základě toho je verifikace buď úspěšná nebo neúspěšná. Níže budou popsány jednotlivé části této metodologie, zejména hierarchie tříd, fázovací systém a jednotlivé komponenty a jejich možnosti. [16] [17]

Hierarchie tříd

UVM poskytuje sadu základních tříd, ze kterých mohou dědit komplexnější třídy. Na obrázku 2.1 je zobrazena základní struktura tříd. Dvě úplně nejzákladnější jsou `UVM_VOID`, která je základnou všech tříd, ale primárně je prázdná, a `UVM_OBJECT`, jenž je hlavní třídou, ve které jsou definovány běžně používané funkce jako je například tisk, kopírování a porovnání dvou objektů stejné třídy. [16]

Existuje 7 základních kategorií, které je potřeba znát, aby bylo možné plně pochopit UVM. Patří mezi ně třídy: [16]

- **UVM Root** – implicitní UVM komponenta nejvyšší úrovně, která je automaticky vytvořena, když se spustí verifikace. Uživatelé k ní mají přístup skrze globální proměnnou, `uvm_top`. Dále se tato třída stará o fázování všech komponent, kontroluje chyby během fáze `end_of_elaboration` a stará se o chyby typu **UVM_FATAL**.

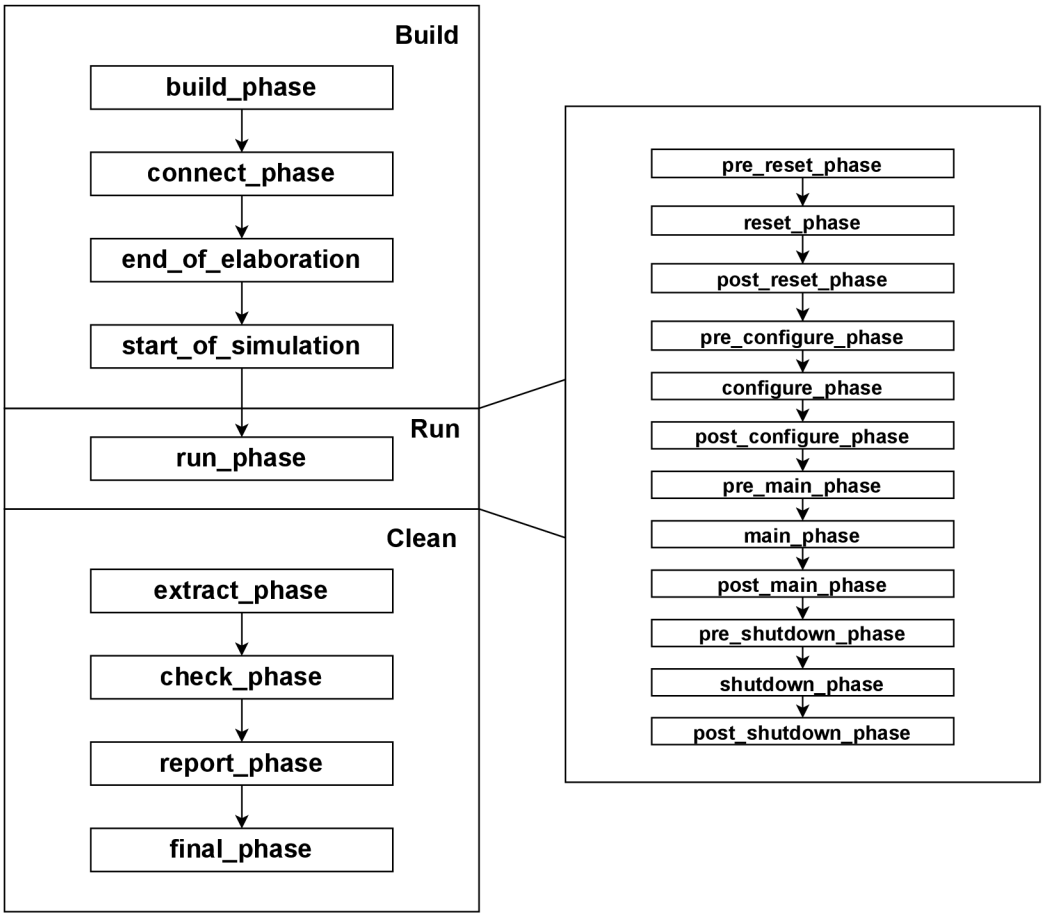


Obr. 2.1: Hierarchie tříd [16]

- **UVM Object** – z této třídy vychází všechny UVM komponenty a transakce. Těmto třídám pak poskytuje běžně používané metody jako je vytváření, kopírování, porovnání, tisk a nahrávání.
- **UVM Report Object** – poskytuje rozhraní pro všechny hlášení. Všechny zprávy, varování, errory prochází skrze toto rozhraní. Obsahuje uvm report handler, který si uchovává nahlašovací konfiguraci na základě, které poskytuje hlášení uživateli. Tato hlášení prochází uvm report serverem, který se stará o generování a formátování zpráv. Zprávy obsahují identifikační údaj, důležitost zprávy, úroveň podrobností a samotnou textovou zprávu.
- **UVM Component** – od této třídy jsou odvozeny veškeré verifikační komponenty a disponuje následujícími rozhraními:
  - **Hierarchie** – poskytuje metody, které se využívají při vyhledávání a procházení hierarchií komponent.
  - **Fázování** – definuje fázovací systém všech komponent. Více v sekci 2.1.1.
  - **Nahlašování** – tato funkcionalita je odvozena ze třídy UVM Report Object.
  - **Záznam** – definuje možnost ukládání transakcí do databáze.
  - **Továrna** – toto rozhraní je používané k vytváření komponent.

2.1.1 Fázovací systém

Fázovací systém, jehož hierarchie je zobrazena na obrázku 2.2, slouží v UVM k synchronizaci životního cyklu simulace. Každá komponenta obsahuje předdefinované fáze, kterými postupně prochází. Synchronizace spočívá v tom, že jakmile verifikace



Obr. 2.2: Fázovací systém [18]

vstoupí do dané fáze, není možné přejít do jiné, dokud všechny komponenty nevykonají kód pro tuto fázi. Pro každou z nich je definovaná metoda, jež je využívaná jako přístupový bod k dané fázi. [17] [18]

UVM Fázovací systém je rozdělen do 3 kategorií. **Build**, která slouží sestavení a spuštění verifikace, obsahuje 4 fáze (funkce):

- **build_phase** – sestavuje komponenty testbenche a vytváří jejich instance.
- **connect_phase** – propojuje různé komponenty testbenche pomocí tzv. TLM portů (slouží k přenosu transakcí skrze různé úrovně hierarchie testbenche).
- **end_of_elaboration_phase** – slouží k vykonání posledních úprav struktury nebo konfigurace testebenche. Mimo jiné je zde možné zobrazit topologii.
- **start_of_simulation_phase** – slouží ke spuštění verifikace. V této části lze nastavit počáteční konfiguraci simulačního běhu nebo případně zobrazit výslednou topologii testbenche.

`Run`, která řídí běh verifikace, obsahuje jako jediná task `run_phase`, která indikuje spuštění simulace. V této fázi dojde ke spuštění testovacích scénářů, započne

odesílání stimulů směrem k DUT a začne probíhat ověřování funkčnosti. Obsahuje další metody stejného typu, které se vykonávají paralelně s `run_phase`. Patří mezi ně `reset`, kde může být upraveno chování verifikace během resetu. `Configure`, jež se stará o chování během konfigurace DUT. `Main`, během které je spuštěn testovací scénář a sekvence. `Shutdown`, signalizuje konec verifikace, čeká se zde než se provedou poslední operace čtení a zápisu na DUT. Ke všem zmíněným metodám existují i metody s prefixem **pre** (může zde být ovlivněno chování, než verifikace vkročí do dané fáze) a **post** (zde lze ovlivnit chování po ukončení dané fáze). [18]

Poslední kategorií je fáze `Clean`, která má za úkol ověřit výsledek verifikace a to tak, že posbírá informace ze scoreboardu a funkčního pokrytí, které využije ke zjištění, zda testovací scénář skončil úspěšně nebo ne. Obsahuje následující metody: [18]

- `extract_phase` – slouží k vytažení výsledků ze scoreboardu a monitorů funkčního pokrytí. Mohou zde být počítány statistické informace jako počet přijatých trasakcí, počet zahozených transakcí atd.
- `check_phase` – tato fáze se využívá k ověřování správnosti chování DUT a k případnému hledání chyb.
- `report_phase` – v této fázi dochází k zobrazení výsledků verifikace, jsou zde využity statistické proměnné z `extract_phase` a jsou vypsány do konzole nebo do souboru.
- `final_phase` – slouží k dokončení posledních akcí, které Testbench nestihl dokončit.

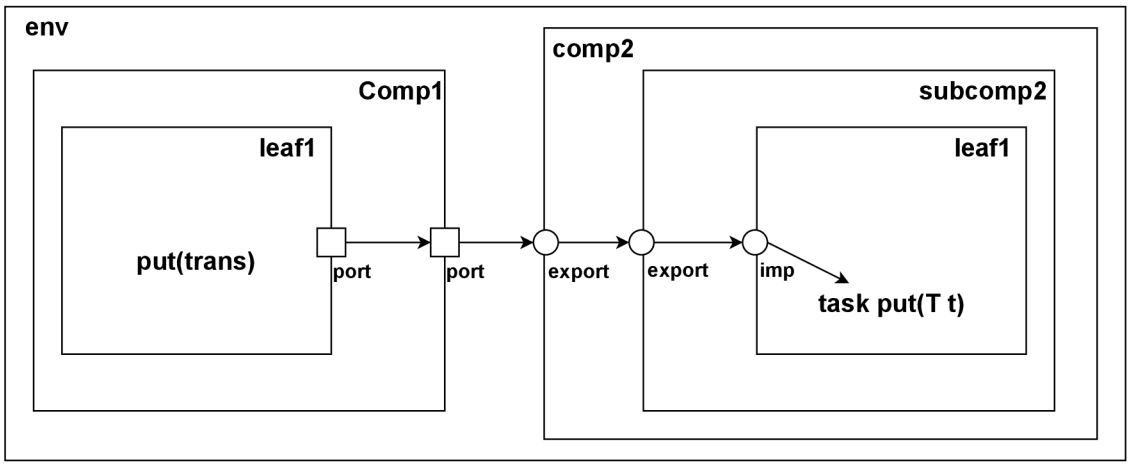
2.1.2 TLM

Před definicí základních komponent je důležité si říct něco o tzv. Transaction Level Modeling (zkráceně TLM). Zjednodušeně se jedná o modelovací styl, který se používá pro vytváření abstraktních objektů a systémů. Data jsou zde reprezentovány tzv. transakcemi, které jsou posílány mezi jednotlivými komponentami skrze různé porty, kterým se říká TLM rozhraní. Tato abstrakce se používá, aby se velké množství signálů verifikovaných systémů nahradilo srozumitelnějším a uchopitelnějším řešením. [16]

UVM disponuje sadou TLM portů, které lze použít k propojení různých komponent, jež jsou použity k odesílání transakcí mezi nimi. Existuje velké množství rozhraní a portů. Porty pak lze rozdělit na blokující a neblokující podle použitých metod a chování při odesílání nebo příjmu transakcí. Dále je můžeme rozdělit na porty, exporty, impy, analysis porty a FIFO. [16] [19]

Porty jsou využity k inicializaci a odesílání transakcí do nejvyšší vrstvy hierarchie, tzn. od podkomponent k hlavní komponentě. Exporty se naopak používají k přijímání a předávání paketů z nejvyšší vrstvy do místa určení (tj. například score-

board), kdy v cíli musí být implementována metoda `put` a musí disponovat analysis impem. U Analysis portů se používá místo metody `put` metoda `write`. Komponenta, která odesílá transakce, volá nad daným portem funkci `write`. Protější strana musí definovat metodu `write` a analysis imp. Poslední variantou je použití propojení pomocí portu typu FIFO. Obě komponenty jsou propojeny pomocí portů a mohou libovolně volat metodu `get` i `put` podle toho, jestli transakce ukládají nebo odebírají z fronty, protože FIFO má obě metody implementované. Použití portů, exportů a impů zobrazuje obrázek 2.3. [16] [19]

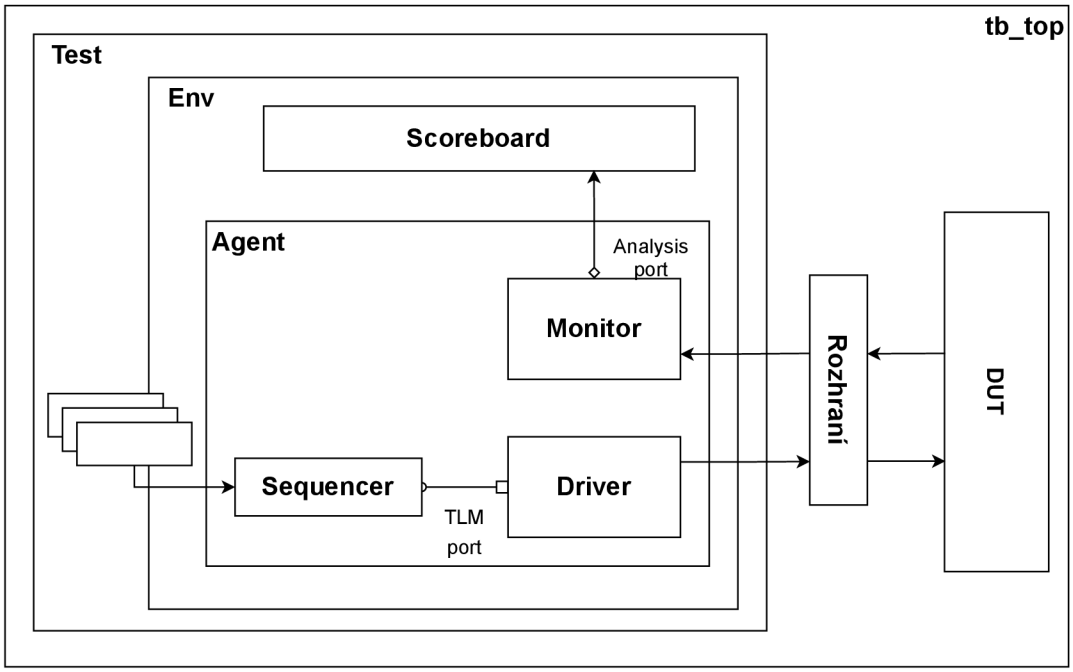


Obr. 2.3: Ukázka TLM portů [19]

Blokující porty vykonají operaci a zablokují se do doby, dokud nepřijde další žádost nebo není třeba odeslat další požadavek. Využívají k tomu metody `get` a `put`. Neblokující porty se nezablokují a stále kontrolují, zda-li neexistuje transakce, která by byla použitelná. Musí být definovaná kontrola, která eliminuje zpracování prázdné transakce. Tento typ používá metody `try_get` a `try_put`. [19]

2.1.3 Komponenty

V UVM jsou jednotlivé části verifikace hierarchicky uspořádané (viz. 2.4) a tvoří ucelenou topologii. Na vrcholu celé hierarchie se nachází tzv. top level modul (testbench), kde jsou vytvořeny instance všech verifikačních komponent, rozhraní a DUT. Následuje test, kde je vytvořena instance prostředí a konfigurační objekt, který je přístupný pro všechny komponenty prostředí skrze databázi. Nejnižší položkou této hierarchie je již zmíněné prostředí, ve kterém se nacházejí jednotlivé verifikační komponenty. Mezi nejdůležitější patří Scoreboard a Agent. Tyto a další komponenty budou popsány níže. [16] [17]



Obr. 2.4: Topologie Testbenche [16]

UVM Testbench Top

Jedná se o statickou schránku, která v sobě obsahuje vše, co je potřeba k sestrojení verifikace, a stává se tak kořenovým uzlem celé hierarchie. Nejčastěji bývá pojmenovaný jako `tb` nebo `tb_top`. Důležitou částí testbenche jsou generátory hodin a resetu. Tak jako reálný systém může využívat různé hodinové domény, i zde musí být definovány generátory hodin, kterými se bude řídit verifikace a samotný systém. Jelikož je často vyžadována škálovatelnost hodinového generátoru, je vhodné vytvořit agenta, aby bylo možné generátor snadněji ovládat sekvencemi a testovacími scénáři. Podobná situace platí i pro generátor resetu, který má za účel navodit různé resetovací scénáře (např. reset na začátku verifikace, reset v průběhu verifikace apod.). V poslední řadě je zde do databáze přidáváno rozhraní DUT, aby k němu měli přístup ostatní komponenty, a probíhá zde spouštění testů. [16] [17]

UVM Test

Test, který dědí třídy `uvm_test`, má na starost tři hlavní funkce: vytvoření instance prostředí, skrze továrnu ho nakonfigurovat a spustit sekvenci, případně knihovnu sekvencí, jež slouží k navození testovacího scénáře. Testů bývá zpravidla větší množství, ale existuje pouze jeden hlavní (obsahuje instanci prostředí a testbenche), ze kterého ostatní dědí. V různých testech pak mohou být provedeny úpravy typu: spuš-

tění jiných sekvencí, změna konfigurace rozhraní atd. Vytváření instancí prostředí a jeho konfigurace se provádí v `build_phase`. Spouštění sekvence je uskutečněno v `run_phase`. Dále je možné si vytisknout topologii celého prostředí v již zmíněné fázi `end_of_elaboration_phase` za pomoci metody `print_topology`, která se dá využít při ladění. Registrace u továrny probíhá opět pomocí makra komponenty. [16] [17]

UVM Environment

Prostředí, jež dědí z třídy `uvm_env`, je objekt obsahující jednotlivé verifikační komponenty nebo i jiná prostředí a jejich konfigurační objekty. Jsou zde definovány například agenti pro různá rozhraní, scoreboard, funkční pokrytí a další. Obsahuje `build_phase`, kde jsou vytvářeny instance jednotlivých komponent prostředí, a `connect_phase`, kde je provedeno jejich propojení. Tak jako všechny komponenty, i prostředí je nutné registrovat u továrny pomocí makra komponenty. [16] [17]

Rozdělení verifikačních komponent do různých prostředí umožňuje lepší znovupoužitelnost verifikačních komponent. Zpravidla může být verifikace rozdělena na vysílací (TX) a přijímací část (RX). Různé části mohou mít různé konfigurace, tudíž je vhodné pro každou z nich definovat různá prostředí. Proč je zapotřebí vytvořit prostředí, když je možné komponenty definovat v testu? Možné to je, ale nedoporučuje se to, protože testy nejsou znovupoužitelné. [16] [17]

UVM Driver

Driver, jenž dědí z třídy `uvm_driver`, je aktivní komponenta, která se stará o převod transakce na signály rozhraní DUT. V `run_phase` driveru je tedy definována logika, kterou je řečeno, jakým způsobem předávat data DUT. Aby mohl driver správně plnit svou funkci, je nutné definovat typ transakce. To je uskutečněno na základě jeho parametru. Driver se taktéž musí registrovat u továrny a to pomocí makra komponenty. [16] [17]

Se sekvencerem je propojen skrze TLM port typu `uvm_seq_item_pull_port`, který umožňuje přijímat žádosti od sekvenceru. Jak již bylo zmíněno, k propojení s DUT se používá rozhraní. Aby mohlo být používáno, je definováno tzv. virtuální rozhraní, do kterého je uložen ukazatel (handle) pomocí konfigurační databáze. Příklad získání ukazatele z databáze je na ukázce 2.1.3, která se nachází v `build_phase` driveru. [16] [17]

```
uvm_config_db#(virtual if_type)::get(this, "", "vif", vif)
```

Komunikace mezi driverem a sekvencerem probíhá tak, že driver v nekonečné smyčce požaduje od sekvenceru transakce, které následně převádí na signály rozhraní

DUT. Po dokončení všech operací odešle zprávu o dokončení sekvenceru a může mu poslat i případnou odpověď. K této komunikaci se používá tzv. `Driver-Sequencer handshake`. K tomu jsou využity následující metody: [16] [17]

- `get_next_item` – blokující metoda (blokuje, dokud není žádost dostupná), která se využívá pro vyžádání transakce od sekvenceru.
- `try_next_item` – neblokující metoda, využívá se ke stejnému účelu jako metoda `get_next_item` s rozdílem, že vrací `null` v případě, že transakce není dostupná, v opačném případě vrací transakci.
- `item_done` – neblokující metoda dokončující handshake mezi driverem a sekvencerem. Obě předchozí metody musí být zakončeny touto metodou.

Modely použití – již zmíněný handshake lze použít několika způsoby: [16] [17]

- **Unidirectional Non-pipelined** – v tomto modelu jsou pouze přijímány a zapisovány data ze sekvenceru a neočekává se žádná odpověď od driveru. V reálu je zavolána metoda `get_next_item` nebo `try_next_item`, následně jsou data zapsána na rozhraní a je zavolána metoda `item_done`.
- **Bidirectional Non-pipelined** – tento model slouží k obousměrnému přenosu, data jsou přijímány ze sekvenceru pomocí stejných metod a zapisovány na rozhraní a zároveň se od driveru očekává odpověď sekvenceru tím, že metodě `item_done` je předložen parametr ve formě odpovědi. Ta je poslána pouze v případě, že v minulosti přišla žádost. Současně může být aktivní pouze jeden přenos.
- **Pipelined** – chováním je podobný předchozímu modelu s tím, že přijetí žádosti se překrývá s odesíláním odpovědi. Výrazně je tím zvýšen výkon, ale i složitost driveru.
- **Out-of-Order Pipelined** – tento model se od ostatních liší tím, že odpověď na žádost může přijít kdykoliv. Každá žádost má svoje ID, na základě kterého je odeslána odpověď.

UVM Sequencer

Sekvencer se stará o generování dat transakcí a jejich odesílání driveru. Speciálním případem je tzv. virtuální sekvencer, který se používá ke sdružení více sekvencerů na jednom místě. Využívá se kvůli zjednodušení správy u větších verifikací. [16] [17]

UVM Sequence_item

`Sequence_item`, nebo také transakce, je objekt, který má za úkol definovat náhodná datová pole pro generovaný stimul. Aby bylo možné transakci používat, je nutné ji zaregistrovat do továrny. Je možné zde nastavit omezení pro jednotlivá datová pole (například hodnoty, kterých dané pole může nabývat a další). Dále jsou zde

definovány pomocné metody, které se využívají napříč komponentami. Nejčastěji se jedná o výpis transakce, porovnání a kopírování. Ty jsou definovány buď pomocí tzv. maker, nebo si je programátor může vytvořit sám. [16] [17]

Pokud si programátor chce zjednodušit práci a metody nechce vytvářet sám, je možné použít tzv. `field makra`, která slouží jako prostředek, kterým je továrně vysvětleno, jakým způsobem dané metody používat. Na ukázce 2.1.3 je znázorněn způsob používání těchto maker. Makra jsou vždy definována mezi klíčovými slovy `_begin` a `_end`, uvnitř následuje definice proměnných, nad kterými budou metody používány. Klíčové slovo `‘uvm_field_` následované datovým typem říká, jakého typu je proměnná, nad kterou se budou metody vykonávat. V našem případě je použitý datový typ `uvm_field_int`. Tento typ mohou používat proměnné typu int, bit a byte. Následně se uvnitř složených závorek nachází dva argumenty `ARG` a `FLAG`. Prvním argumentem je předána proměnná a jako druhý argumentem je určeno, jaké metody mohou být použity nad danou proměnnou. `FLAG` může nabývat hodnot: [16] [17]

- `UVM_ALL_ON` – všechny operace jsou povoleny.
- `UVM_DEFAULT` – má stejnou funkci jako předchozí flag.
- `UVM_NOCOPY` – nad danou proměnnou není povolena metoda kopírování.
- `UVM_NOCOMPARE` – nad danou proměnnou není povolena metoda porovnávání.
- `UVM_NOPRINT` – nad danou proměnnou není povolena metoda výpisu.
- `UVM_NOPACK` – nad danou proměnnou není povolena metoda rozbalení nebo zabalení.

```
‘uvm_object_utils_begin(Frame_Macro)
    ‘uvm_field_int(m_data, UVM_DEFAULT)
    ‘uvm_field_int(m_byte_enable, UVM_DEFAULT)
    ‘uvm_field_int(m_error_cnt, UVM_DEFAULT)
‘uvm_object_utils_end
```

UVM Sequence

Sekvence je objekt, který dědí ze třídy `uvm_sequence` a stará se o generování náhodné sady transakcí, které skrze sekvencer posílá driveru. Ve své podstatě definuje, jak budou vypadat transakce, které následně driver zapíše ve formě signálů na rozhraní DUT. Aby mohla provádět svou činnost, potřebuje znát typ transakce, který je jí předán formou parametru. Sestává ze dvou částí, konstruktoru, kde je pouze zavolán konstruktor předka pomocí klíčového slova `super`, a body, task, ve kterém je umístěna logika sekvence (vytvoření sekvence a odeslání driveru). U továrny se registruje pomocí makra objektů. [16] [17]

Logika, která se nachází uvnitř `body` se dá rozdělit do 6 fází: [16] [17]

- **Vytvoření transakce** – vytvoří `sequence_item` pomocí metody `create`.
- **Vyčkání na požadavek** – v tomto kroku se čeká až bude v driveru zavolána metoda `get_next_item`.
- **Randomizace** – v té to fázi dojde k vygenerování náhodných dat transakce. Při generování mohou být definována omezení, která mohou například stanovit, že nějaký signál musí být nastavený do logické 1.
- **Odeslání žádosti** – odešle žádost sekvenceru, který ji přepošle driveru.
- **Čekání na dokončení** – volitelný krok, kde se task `body` zablokuje do doby, než driver zavolá `item_done`.
- **Obdržení odpovědi** – opět volitelný krok, používá se pokud se očekává odpověď od driveru.

UVM Monitor

Monitor, jenž dědí z třídy `uvm_monitor`, je pasivní komponenta, která má za úkol zachycovat změny signálů na rozhraní systému a převádět je na transakce, které mohou být následně dál zpracovávány. Dále je odpovědný za kontrolu správného chování sběrnice (celistvost paketu nebo kontrola mezipaketových mezer) a funkčního pokrytí. Také obsahuje virtuální rozhraní, které je opět vytažené z databáze v `build_phase`, navíc obsahuje tzv. TLM Analysis port (instance je vytvořena v `build_phase`), kterým odesílá transakce dalším komponentám. Nejdůležitější částí monitoru je `run_phase`, kde je definována jeho funkcionalita, a je zde zapisováno na TLM Analysis port pomocí metody `Write`. [16] [17]

UVM Agent

Agent, jenž dědí ze třídy `uvm_agent`, je komponenta, která má za úkol zapouzdřit sekvencer, driver a monitor do jedné entity. Ty jsou mezi sebou propojeny pomocí TLM rozhraní. Hlavní funkcí agenta je definovat metody pro generování transakcí, monitorování rozhraní a kontrolu pokrytí na základě specifikací protokolu sběrnice. U továrny je opět registrovaný pomocí makra komponenty. Vytvoření instancí komponent se provádí v `build_phase`. Existují dva typy agentů: [16] [17]

- **Aktivní** – disponuje všemi třemi komponentami, tudíž umožňuje zápis dat na sběrnici a zároveň její monitorování.
- **Pasivní** – obsahuje pouze monitor, tudíž umožňuje pouze monitorování provozu. Nejčastěji se používá v případech, kdy se sběrnice nachází uprostřed DUT a my chceme monitorovat jeho vnitřní stav.

Skutečnost, zda-li je agent pasivní nebo aktivní, je určena na základě metody `get_is_active`, která vrací typ agenta, ten musí být nakonfigurován pomocí konfiguračního objektu agenta. Pokud je výstup této metody `UVM_ACTIVE`, je v `connect_phase`

provedeno propojení driveru se sekvencerem. [16] [17]

UVM Scoreboard

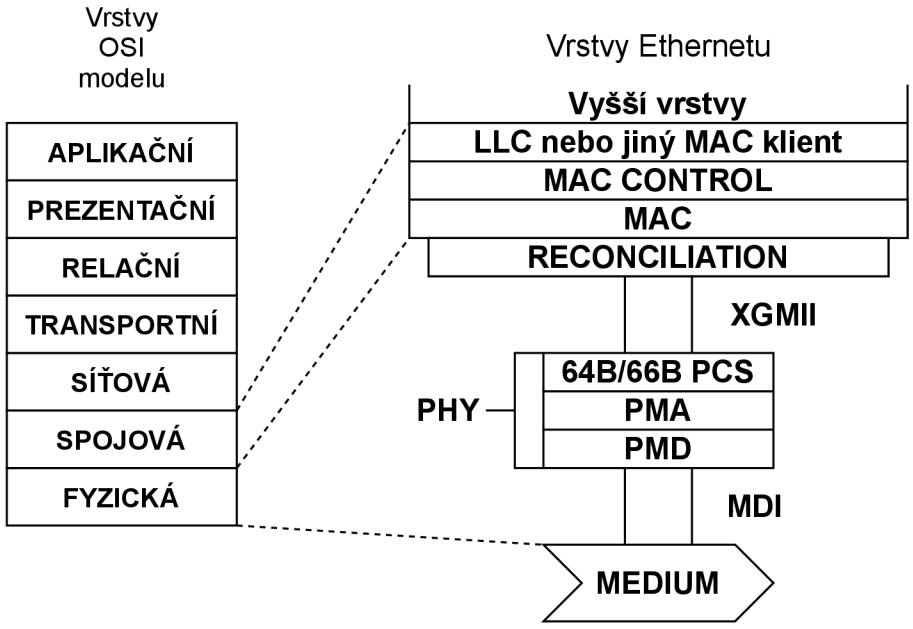
Verifikační komponenta, která má za úkol porovnat výstupní transakci DUT s referenční transakcí, která byla vytvořena v tzv. referenčním modelu. Po vyhodnocení všech transakcí je rozhodnuto, zda verifikace proběhla úspěšně, nebo ne. Taktéž musí být registrován u továrny pomocí makra komponenty. Ve scoreboardu jsou definovány tzv. TLM Analysis porty, které slouží k zachycování výstupních transakcí z rozhraní. Dále je definováno tzv. TLM Analysis FIFO, do kterého jsou zapisovány vstupní transakce. Jak FIFO, tak porty disponují metodou `Write`, kterou používají monitory. Aby bylo možné s nimi pracovat, je nutné vytvořit jejich instance v `build_phase`. [16] [17]

Jak již bylo zmíněno, ve scoreboardu se nachází referenční model, který přebírá stejné transakce jako DUT, následně nad nimi provádí úpravy odpovídající specifikaci systému, tím vznikne referenční transakce, která je poslána do scoreboardu k porovnání skrze TLM Analysis port. [16] [17]

3 Ethernet

Ethernet je technologie umožňující propojení zařízení v kabelové LAN (Local Area Network) a WAN (Wide Area Network). Definuje protokol, který vznikl v roce 1985 pod institutem `IEEE`, dokument vystupuje pod jménem `IEEE 802.3` a existuje v různých verzích. Jsou v něm popsány pravidla pro různé technologie od 10BASE5 (technologie přenášející data po koaxiálním kabelem rychlostí 10 Mbit/s) po 400GBASE-R (technologie pro přenos dat po optickém kabelu rychlostí 400 Gbit/s). Protokol pro ně popisuje, jakým způsobem se mají data přenášet, jejich formát, jaké přenosové médium k tomu použít a jiné další funkcionality, které budou popsány v sekcích níže. Tato kapitola bude zaměřena na technologii `10GBASE-R` a bude popsána její funkcionalita. [20] [21]

Z pohledu ISO/OSI modelu můžeme Ethernet začlenit do dvou vrstev, spojové a fyzické viz. obrázek 3.1. Spojovou vrstvu můžeme dále rozčlenit na vrstvu LLC (LogicaL Link Control) a MAC (Media Access Control). Fyzická vrstva se dále člení na vrstvy PCS (Physical Coding Sublayer), PMA (Physical Medium Attachment) a PMD (Physical Medium Dependent). Rozhraní mezi fyzickou a MAC vrstvou tvoří tzv. Reconciliation Sublayer (RS) spolu s XGMII rozhraním. RS tedy umožňuje mapování dat z vrstvy na jednotlivé signály XGMII rozhraní, po němž jsou posléze data posílány na nižší vrstvy. [27]



Obr. 3.1: Architektura 10 Gb Ethernetu [27]

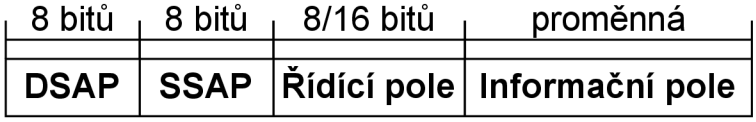
3.1 Spojová vrstva

Spojová vrstva je druhá vrstva OSI modelu, jejímž cílem je zajistit přenos dat mezi sousedními síťovými uzly v LAN nebo WAN sítích. Je zodpovědná za převod datových toků vyšších vrstev na signály, se kterými dokáže pracovat fyzická vrstva. Disponuje různými funkcemi. První z nich je zapouzdřování paketů síťové vrstvy do tzv. rámců, aby bylo možné s nimi pracovat na této vrstvě. Druhou důležitou funkcí je adresování, definuje tedy adresy, které identifikují zařízení, ze kterého je generován provoz. Třetí funkcí je synchronizace, aby nedocházelo k chybám v přenosu mezi dvěma zařízeními je nutné tyto zařízení sladit tak, aby se nacházela ve stejném stavu. Poslední funkcí, která bude zmíněna, je kontrola chyb. Při přenosu dat mezi dvěma zařízeními může docházet k chybám, proto je nutné definovat způsob, jak ověřit správnost přeneseného rámce. Na této vrstvě se k tomu používá CRC (Cyclic Redundancy Check). Podvrstvy, které byly zmíněné v předchozí kapitole budou popsány níže. [22]

3.1.1 Logical Link Control (LLC)

LLC se stará o komunikaci mezi vyššími a nižšími vrstvami, v tomto případě se jedná o spojovou a síťovou vrstvu. Přebírá data síťového protokolu, typicky IPv4 nebo IPv6 pakety, a připojuje k nim kontrolní informace, které pomáhají s doručením paketu cílovému uzlu. LLC se implementuje v softwaru a nezávisí na fyzické vrstvě. Můžeme si ji představit jako ovladač pro síťovou kartu (NIC), který pomocí přímé komunikace s kartou předává její data MAC vrstvě. Na této vrstvě jsou používány jednotky nesoucí název LLC PDU (Protocol Data Unit). Jak je patrné z obrázku 3.2, obsahuje následující pole: [23] [24]

- **DSAP** – obsahuje IP adresu příjemce.
- **SSAP** – obsahuje IP adresu odesílatele.
- **Řídící pole** – obsahuje informace, které jsou používány k řízení toku data a kontrole chybovosti. Může obsahovat sekvenční číslo paketu, sekvenční číslo potvrzovaného paketu a stav přijímače, které může nabývat hodnot `RNR` (nepřipraven přijímat), `RR` (připraven přijímat) a `REJ` (zamítnuto).
- **Informační pole** – obsahuje nejčastěji data síťové vrstvy.

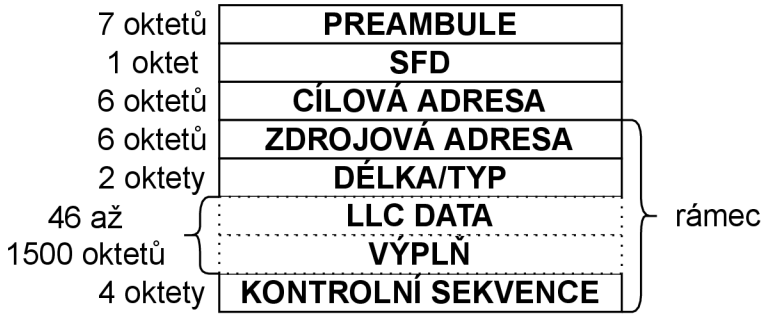


Obr. 3.2: LLC PDU [24]

3.1.2 Media Access Control (MAC)

Druhou podvrstvou spojové vrstvy je MAC, která umožňuje více zařízením sdílet jeden komunikační kanál. Aby to bylo možné, existují na této vrstvě metody pro přístup k médiu. Úkolem této funkcionality je umisťování rámců na přenosový kanál tak, aby nedocházelo ke kolizím s jinými zařízeními. Této metodě se říká CSMA/CD. MAC lze použít ve dvou módech, `half-duplex`, kdy může najednou komunikovat jen jedno zařízení, kde je právě využito techniky CSMA/CD, a `full-duplex`, kde se předpokládá, že mohou komunikovat obě zařízení najednou. Mimo jiné se tato vrstva stará o zapouzdřování dat. První důležitou operací, kterou tato funkcionalita má za úkol, je vytváření rámců, jejich synchronizace připojením tzv. preambule a určení hranic rámců (začátek a konec). Další neméně důležitou operací je adresování pomocí tzv. MAC adres, které jsou identifikátory odesílatele a příjemce. Poslední operací je detekce chyb, tím je chápána kontrola bezchybnosti přenosu pomocí CRC. V sekcích níže bude podrobněji popsáno vysílání a příjem rámců a jejich formát. [26] [25] [23]

Formát rámce



Obr. 3.3: Ethernetový rámec [26]

Na obrázku 3.3 je vyobrazen ethernetový rámec, který je vytvořen MAC vrstvou. V této sekci bude popsána jeho struktura a některé operace, které jsou při vytváření jednotlivých položek potřeba vykonat. Položkou, kterou rámec začíná je `preambule`, tato položka má 7 oktetů a slouží ke stabilizaci a synchronizaci přenosového média. Preambule má následující tvar:

```
10101010 10101010 10101010 10101010 10101010 10101010 10101010
```

Její bity jsou přenášeny zleva doprava. Důvodem takovéto posloupnosti je, že kódování Manchester, které bylo dříve využíváno na fyzické vrstvě, se jeví jako periodicky se opakující, a tudíž je vhodná pro synchronizaci. Následuje tzv. Start Frame

Delimiter (`SFD`), který slouží jako identifikátor začátku rámce, vypadá následovně: `10101011`. Další položkou je adresa příjemce, která slouží jako identifikátor stanice, které je rámec odesílán, a odesílatele, která slouží jako identifikátor stanice, která rámec vytváří a odesílá. Používá se tzv. MAC adresa, která musí mít délku 48 bitů, obsahuje následující položky: [26] [25]

- **I/G** – toto pole určuje o jaký typ adresy se jedná. Pokud toto pole obsahuje logickou 0, jedná se o tzv. individuální adresu (adresa stanice). V případě logické 1 se jedná o tzv. skupinovou adresu, tj. adresa patřící skupině stanic.
- **U/L** – tímto polem je určeno, zda se jedná o adresu globální (veřejná adresa, která se používá v internetu) nebo lokální (adresa používaná v lokální síti).
- **48bitová adresa**

Dalším polem je **Délka/typ**, kde mohou být uloženy dva druhy záznamů. Pokud délka rámce dosahuje maximálně hodnoty 1500 dekadicky, tak je zde uložena velikost datové části v bytech. V případě, že hodnota tohoto pole je větší jak 1536 dekadicky, je zde uložen typ protokolu. Dále se zde nachází datová část, zde se nachází data LLC nebo jiného MAC klienta. Za daty se nachází tzv. výplň, která se odvíjí od položky délka/typ. Pokud je hodnota této položky menší než minimální velikost ethernetového rámce, do tohoto pole se uloží takový počet oktetů, aby délka byla minimálně 64 bytů (minimální velikost ethernetového rámce). [27]

Poslední položkou ethernetové rámce je Kontrolní sekvence (**FCS**). Pro její výpočet se používá CRC a počítá se z celého rámce. Ve své podstatě se jedná o kód pro detekci chyb. Pro kódování je definovaný generační polynom, který je pro každou velikost CRC jiný. V našem případě nás bude zajímat CRC32, jehož velikost je právě 32 bitů a generační polynom má tvar: $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ Matematicky je CRC definováno následující procedurou: [26] [25]

- První bity rámce jsou doplněny o 32 bitů logických 0.
- Výsledné bity rámce jsou pak považovány za koeficienty polynomu $M(x)$ stupně n-1, kde n je počet bitů rámce. Prvky jsou uspořádány tak, že první prvek polynomu odpovídá prvnímu bitu cílové adresy a poslední náleží buď poslednímu bitu dat, nebo výplně.
- Tento polynom je v dalším kroku vynásoben členem x^{32} a podělen generačním polynomem. Těmito operacemi vznikne zbytek $R(x)$ 31. stupně. Na závěr je bitová sekvence polynomu$M(x)$ doplněna logickými 0 a zbytek $R(x)$ je prohlášen za hodnotu CRC. Tato hodnota je následně vložena do pole FCS, které je pak porovnáno s hodnotou CRC, kterou si vypočítá protistrana z přijatého paketu.

Vysílání rámce

Pokud je obdržena žádost na odeslání rámce LLC vrstvou, započne vytváření rámce. S žádostí MAC vrstva obdrží i potřebné informace, jako je adresa odesílatele a příjemce, data a případné dodatečné kontrolní data. MAC tyto informace zabalí do datového bloku, předřadí k nim tzv. preambuli, označení začátku rámce (SFD), MAC adresu odesílatele a příjemce a položku obsahující délku rámce, případně typ zapouzdřeného protokolu. Dalším krokem je ze získaných informacích o rámci (délka/typ) ověřit, zda-li délka odesílaného rámce odpovídá alespoň minimální délce, a pokud tomu tak není, je doplněna tzv. výplň. Na závěr je za pomoci CRC vypočítána hodnota hodnota FCS, která umožňuje ověřit bezchybnost rámce. FCS musí být počítáno pouze v případě, že ho MAC vrstva neobdržela od LLC vrstvy. Následně, pokud je vytváření rámce dokončeno a jedná se o `half-duplex` přenos, je odeslán ke zpracování komponentě Media Access Management. Tato komponenta zkontroluje, zda někdo na sdíleném kanálu nevysílá. Pokud vysílá, počká daný časový interval a zkusí dostupnost média znovu. Pokud nikdo na kanálu nevysílá, je rámec odeslán fyzické vrstvě k dalšímu zpracování. Pokud se jedná o mód `full-duplex`, je rámec po vytvoření odeslán fyzické vrstvě. Důležitým aspektem je, že vysílač po odeslání rámce čeká určitou dobu, aby bylo možné srovnat drobné rozdíly v přenosových rychlostech mezi vysílačem a přijímačem, a až poté může odeslat další rámec. Této pauze se říká mezipaketová mezera, její hodnota je různá pro každou technologii, například pro technologii 10GBASE-R je tato hodnota minimálně 9,6 ns. [26]

Příjem rámce

Příjem rámce prvně detekuje fyzická vrstva, která se synchronizuje s příchozí preambulí. Následně jsou tato data opět převedena do binární formy a odeslána MAC vrstvě k dalšímu zpracování. První operací po přijetí rámce touto vrstvou je odebrání preambule a SFD indikátoru. V dalším kroku je na základě cílové adresy zkontrolováno, zda data náleží dané stanici. Pokud je stanice příjemcem, jsou položky cílová adresa, zdrojová adresa, délka/typ, data a volitelně FCS odeslány LLC vrstvě nebo jinému MAC klientovi spolu s adekvátním stavovým kódem. Při příjmu rámce MAC vrstvou je také ověřena jeho bezchybnost porovnáním políčka FCS s hodnotou CRC, kterou vypočítala přijímací stanice. Na závěr je provedena kontrola zarovnání rámce. [26]

3.2 Fyzická vrstva

Fyzická vrstva je nejnižší vrstvou OSI modelu, která jako jediná komunikuje s fyzickým hardwarem. Má za úkol přijmout rámce spojové vrstvy a převést je na elek-

trické signály, které mají formu binární dat. Následně zajišťuje přenos po drátovém či bezdrátovém médiu. Pro úspěšné vykonání tohoto procesu jsou definovány různé parametry a procedury, mezi které patří například: [27] [23]

- **Kódování** – určení způsobu, kterým budou bity převedeny na signál. Nejčastěji pomocí tzv. linkových kódů.
- **Přenosová rychlost** – na základě použité technologie je stanovena přenosová rychlost.
- **Synchronizace** – určuje způsob, kterým je provedeno sladění vysílače s přijímačem, aby mohl být proveden přenos. Nejčastěji se využívá kódování dat pomocí již zmíněných linkových kódů, které definují synchronizační mechanismy, například `Manchester`.
- **Přenosový režim** – na této úrovni je rozhodnuto, zda-li bude použitý již zmíněný `half-duplex` nebo `full-duplex`.
- **Definice přenosového média** – definuje, jakým způsobem budou zařízení mezi sebou propojeny.

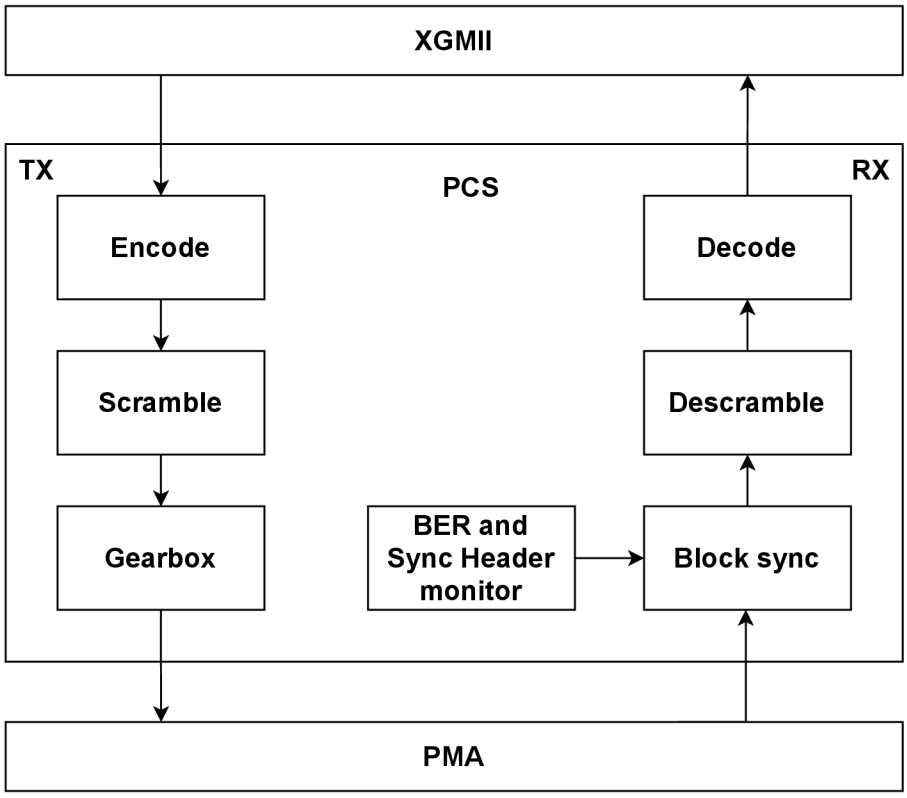
Jelikož se Diplomová práce zaobírá problematikou 10 Gb Ethernetu, bude popisována fyzická vrstva této technologie. Jak již bylo zmíněno v kapitole 3, tato vrstva sestává z 3 podvrstev PCS, PMA a PMD, jejichž funkcionalita bude popsána níže. Největší důraz bude kladen na podvrstvu PCS, protože je součástí verifikované komponenty. [27]

3.2.1 Physical Coding Sublayer (PCS)

Podvrstva `PCS` je nejvyšší podvrstvou fyzické vrstvy, která sídlí mezi `MAC` a `PMA` podvrstvou. Její funkcionalitu lze rozdělit na podprocesy, vysílání, bloková synchronizace, příjem a kontrolu chybovosti pomocí `BER` monitoru. Pokud `PCS` komunikuje s MAC vrstvou, využívá rozhraní `XGMII`, kterým prochází 32bitová datová a 4bitová kontrolní slova (slouží k určení pozice bytu, který je kontrolní). Tato data jsou dále zarovnávána do bloků širokých 66 bitů (64 bitů data a 2 bity synchronizační hlavička), toto uspořádání je označeno jako kódování 64B/66B a bude popsáno níže. Tato data jsou posléze pomocí bloku `Gearbox` převedena na 16bitová slova a odeslána po synchronní 16bitové sběrnici. Funkcionalita této vrstvy bude rozdělena na vysílací a přijímací část a bude popsána v sekcích níže. Blokové schéma této podvrstvy je zobrazeno na 3.4. [27]

Vysílací část

Na vysílací straně jsou přebírány a dále zpracovány datové rámce od MAC vrstvy. Během vysílacího procesu jsou nejprve přijaté rámce sloučeny po dvou, aby vznikla 64bitová slova, a na základě kontrolních 4bitových slov je určeno, zda-li se jedná



Obr. 3.4: Struktura PCS [27]

o kontrolní nebo datový rámec. Pokud je alespoň jeden bit 4bitového kontrolního slova nastaven do logické 1 (1 byte rámce je kontrolní), jedná se o rámec obsahující kontrolní data. Takovému slovu je přidělena synchronizační hlavička `10`. Pokud není ani jeden byte tohoto slova kontrolní, jedná o datový rámec, tudíž je mu přidělena synchronizační hlavička `01`. Takto připravená data jsou posléze zakódovaná. Dosud zmíněné operace provádí blok `Encode`. Zakódovaná data bez hlaviček jsou následně scramblována blokem `Scramble` a převedena blokem `Gearbox` na 16bitová slova. Takto upravená data jsou odeslána podvrstvě PMA. Dále bude popsána funkcionalita již zmíněných bloků vysílací strany PCS podvrstvy. [27]

Blok `Encode` má za úkol zakódovat přijatá data do 66bitových bloků. Tato funkcionalita je použita pro zlepšení přenosových charakteristik dat přenášených po lince a aby bylo možné od sebe oddělit datová a řídící slova. Kódováním zajistíme, že v datovém toku fyzické vrstvy bude dostatek přechodů pro obnovu hodinové domény na straně přijímače. V poslední řadě jsou zde synchronizační hlavičky, které identifikují zarovnání bloků, které je užitečné pro přijímač. Nejčastěji se používá právě kódování 64B/66B. [27]

Proces kódování probíhá následujícím způsobem: Po přiřazení synchronizačních hlaviček a zarovnání do 64bitových bloků jsou data zakódovaná na základě

Název bloku	Označení	XGMII kontrolní kód	10GBASE-R kontrolní kód
idle	/I/	0x07	0x00
LPI	/LI/	0x06	0x06
start	/S/	0xfb	Zakódováno blokem Encode
terminate	/T/	0xfd	Zakódováno blokem Encode
error	/E/	0xfe	0x1e
Sequence ordered set	/Q/	0x9c	Zakódováno blokem Encode
reserved0	/R/	0x1c	0x2d
reserved1		0x3c	0x33
reserved2	/A/	0x7c	0x4b
reserved3	/K/	0xbc	0x55
reserved4		0xdc	0x66
reserved5		0xf7	0x78
Signal ordered set	/Fsig/	0x5c	Zakódováno blokem Encode

Tab. 3.1: Tabulka kontrolních kódů [27]

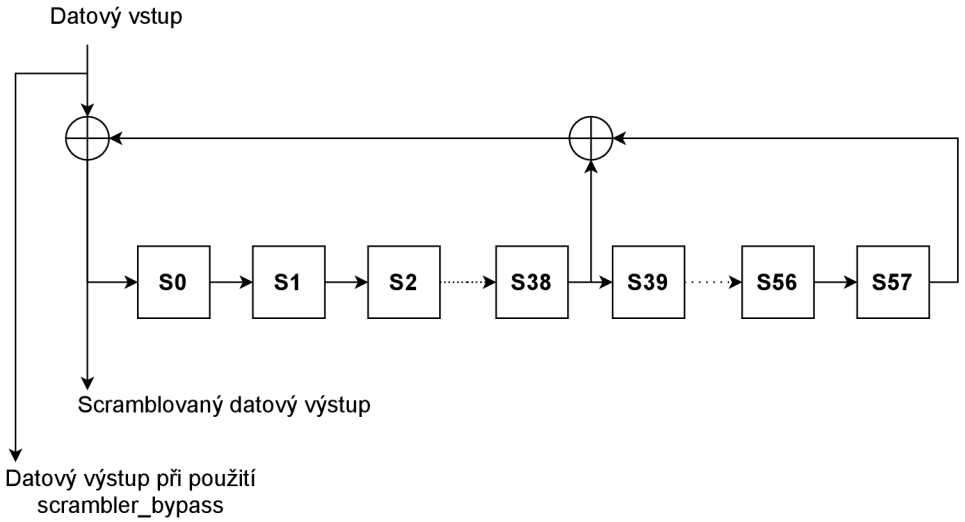
jejich struktury podle tabulky, která je patrná z obrázku 3.5. Přijatá kontrolní data od MAC vrstvy vždy obsahují kontrolní kód a pozici bytu (již zmíněná kontrolní data), na kterém se nachází. Kontrolní kódy jsou poznačeny v tabulce 3.1. Prvním typem dat, které se mohou dostat na vstup komponenty Encode jsou mezipaketové mezery, tato skutečnost je zakódovaná jako tzv. `idle` sekvence. Ta obsahuje na pozici prvního bytu typ bloku s hexadecimální označením `0x1e` a na následujících pozicích kontrolní data (7bitové bloky $C_0 - C_7$). Dalšími možnými vstupem kodéru jsou stavová data, která jsou kodérem označována jako `Ordered sets`. Mohou mít různé kontrolní kódy, jež jsou závislé na obsahu dat, která přenáší. V tabulce kontrolních kódů 3.1 jsou označeny jako `reserved`, `Signal ordered set` a nebo `Sequence ordered set`. Nejdůležitějším blokem je `Sequence ordered set`, který je odesílán při pádu linky. V literatuře je také označován jako `local fault`. Typ bloku těchto sekvencí, označený v tabulce 3.5 písmenkem O, musí být umístěn vždy v prvním oktetu 32bitového slova a může mít různé typové označení. Další možností je, že se na vstupu kodéru objeví první datový blok, který se skládá z preambule a identifikátoru SFD. Takovýto blok je zakódován jako sekvence `start` a může mu být přiřazen typ `0x78`, pokud se jedná o prvních 32 bitů 64bitového bloku a nebo `0x33`,

Input Data	Sync	Block Payload							
Bit Position: / Data Block Format:	0 1	2							65
D_0 D_1 D_2 D_3/D_4 D_5 D_6 D_7	01	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
Control Block Formats:		**Block Type Field**							
C_0 C_1 C_2 C_3/C_4 C_5 C_6 C_7	10	0x1e	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
C_0 C_1 C_2 C_3/O_4 D_5 D_6 D_7	10	0x2d	C_0	C_1	C_2	C_3	O_4	D_5	D_6	D_7
C_0 C_1 C_2 C_3/S_4 D_5 D_6 D_7	10	0x33	C_0	C_1	C_2	C_3		D_5	D_6	D_7
O_0 D_1 D_2 D_3/S_4 D_5 D_6 D_7	10	0x66	D_1	D_2	D_3	O_0		D_5	D_6	D_7
O_0 D_1 D_2 D_3/O_4 D_5 D_6 D_7	10	0x55	D_1	D_2	D_3	O_0	O_4	D_5	D_6	D_7
S_0 D_1 D_2 D_3/D_4 D_5 D_6 D_7	10	0x78	D_1	D_2	D_3	D_4		D_5	D_6	D_7
O_0 D_1 D_2 D_3/C_4 C_5 C_6 C_7	10	0x4b	D_1	D_2	D_3	O_0	C_4	C_5	C_6	C_7
T_0 C_1 C_2 C_3/C_4 C_5 C_6 C_7	10	0x87		C_1	C_2	C_3	C_4	C_5	C_6	C_7
D_0 T_1 C_2 C_3/C_4 C_5 C_6 C_7	10	0x99	D_0		C_2	C_3	C_4	C_5	C_6	C_7
D_0 D_1 T_2 C_3/C_4 C_5 C_6 C_7	10	0xaa	D_0	D_1		C_3	C_4	C_5	C_6	C_7
D_0 D_1 D_2 T_3/C_4 C_5 C_6 C_7	10	0xb4	D_0	D_1	D_2		C_4	C_5	C_6	C_7
D_0 D_1 D_2 D_3/T_4 C_5 C_6 C_7	10	0xcc	D_0	D_1	D_2	D_3		C_5	C_6	C_7
D_0 D_1 D_2 D_3/D_4 T_5 C_6 C_7	10	0xd2	D_0	D_1	D_2	D_3	D_4		C_6	C_7
D_0 D_1 D_2 D_3/D_4 D_5 T_6 C_7	10	0xe1	D_0	D_1	D_2	D_3	D_4	D_5		C_7
D_0 D_1 D_2 D_3/D_4 D_5 D_6 T_7	10	0xff	D_0	D_1	D_2	D_3	D_4	D_5	D_6	

Obr. 3.5: Tabulka typů [27]

když jde o druhou část 64bitového bloku. Dalším typem vstupu kodéru jsou poslední data nacházející se v rámci, označován též jako `terminate`. Tento blok je v tabulce 3.1 označován písmenem `T` a může být kódován různým způsobem a to na základě zarovnání konce dat. Umístění typu bloku a jeho označení je patrné z tabulky 3.5. Posledním kontrolním blokem je tzv. `error`. Ten je odeslán, pokud je detekován vadný blok. Díky tomu je umožněno informovat příjemce, že blok obsahuje chybná data. Takovému bloku kodér přidělí typ bloku, který je hexadecimálně označován `0x1e`. [27]

Proces scramblování se na této vrstvě provádí za pomoci samosynchronizujícího se scrambleru, který ke scramblování využívá generační polynom: $G(x) = 1 + x^{39} + x^{58}$. Funkční schéma tohoto bloku je zobrazeno na obr. 3.6. Na vstup scrambleru je přivedený zakódovaný datový tok bitů bez hlavičky, nad kterým je prováděna operace `XOR` s již zmíněným generačním polynomem. Takto scramblovaná data jsou odesílána na výstup. Pokud je to nutné, lze scrambler vypnout pomocí signálu `scrambler_bypass`, který, jak z překladu vyplývá, vytváří objížďku. Scramblování je tak jako kódování používáno ke zvýšení hustoty přechodů v linkovém signálu. [27]



Obr. 3.6: Funkcionální schéma scrambleru [27]

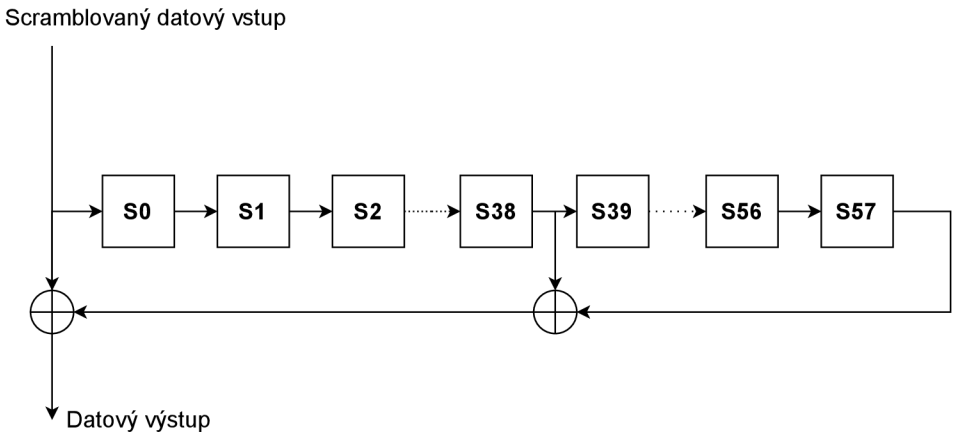
Přijímací část

Prvním úkolem přijímače je průběžně přebírat 16bitová datová slova od podvrstvy PMA a převádět je na 66bitové bloky. Následuje synchronizace na začátek bloku, tuto funkcionalitu poskytuje blok Block sync. K této kontrole se používá tzv. Sync Header monitor a to tak, že na základě 2bitových hlaviček je zjišťováno, zda-li se nachází přijímač na začátku bloku. Nejprve je zkontrolována správnost prvních 64 bloků, v případě bezchybnosti je nastaven signál `block_lock` do logické 1, který signalizuje synchronizaci přijímače na začátek bloku. Block sync provádí kontrolu i po synchronizaci a to tak, že pokud při kontrole narazí na více jak 16 ze 64 vadných hlaviček, prohlašuje, že PCS ztratila synchronizaci a nastavuje signál `block_lock` do logické 0. Podobnou funkci má BER monitor, ten ale tímto způsobem zjišťuje chybovost linky. V případě velké chybovosti nastavuje signál `link_status` do logické 0 na 125μs. Po prvotní kontrole jsou dále data posílána bloku `Descramble`. Ten má za úkol descramblovat nečitelná data na zakódované datové bloky. Ty jsou následně poslány dekodéru, který je dekóduje, rozdělí na 32bitové datové rámce a odešle MAC vrstvě ke zpracování. [27]

Descramblování je proces převedení nečitelných vstupních dat zpět na 64bitové zakódované bloky. Operace je podobná scramblování s tím, že na vstup jsou přivedena scramblovaná data. Ta jsou po dokončení operace odeslána dekodéru k dalšímu zpracování. Funkční schéma descrambleru je patrné z obr. 3.7. [27]

Proces Dekódování umožňuje přijatá descramblovaná data převést opět do datových rámců. Dekódování je prováděno obdobným způsobem jako kódování, na základě tabulky 3.5. Správně dekódovaná data opět obsahují kontrolní kód odpoví-

dající danému bloku, viz. tabulka 3.1. Tato slova jsou následně opět rozdělena na 32bitové datová slova a odeslána MAC vrstvě k dalšímu zpracování. [27]



Obr. 3.7: Funkcionální schéma descrambleru [27]

3.2.2 Physical Medium Attachment (PMA)

PMA je prostřední podvrstva fyzické vrstvy, která definuje servisní rozhraní pro podvrstvu PCS a poskytuje ji následující funkce: [27]

- **Vysílací směr (PMA -> PMD):**
  - Generování hodinového signálu PMA klienta.
  - Serializace datových slov obdržených od PCS podvrstvy.
  - V poslední řadě odesílání serializovaných data podvrstvě PMD.
- **Přijímací směr (PMD -> PMA):**
  - Proces zotavení hodinové domény serializovaných dat podvrstvy PMD.
  - Generování hodinového signálu PMA klienta.
  - Deserializace příchozích dat z PMD podvrstvy.
  - V poslední řadě odesílání deserializovaných dat PMA klientovi a poskytnutí informací o stavu linky pomocí signálu `link status`.

3.2.3 Physical medium dependent (PMD)

Poslední podvrstva fyzické vrstvy definuje servisní rozhraní, které má na straně vysílače za úkol uskutečnit přenos serializovaných dat z PMA vrstvy na přenosové médium. Aby to bylo možné, musí být tato data převedena na signálové toky vhodné pro dané přenosové médium. Na straně přijímače jsou naopak tyto signálové toky převáděny zpět na serializovaná data a jsou odeslány vyšší PMA podvrstvě k dalšímu zpracování. [27]

3.3 Rozhraní XGMII

Komunikační rozhraní používané pro technologii 10GBASE-R, které má za úkol zajišťovat propojení mezi MAC podvrstvou a fyzickou vrstvou (PHY). `XGMII` rozhraní obsahuje 3 typy signálů: datový, řídící a hodinový. Datový signál, označován jako `TXD` na vysílací a `RXD` na přijímací straně, se skládá ze 4 linek, kdy každá obsahuje 8bitový blok. Struktura odeslaného rámce je následující: [27]

```
<inter-frame><preamble><sfd><data><efd>
```

Popis jednotlivých polí je již znám ze sekce 3.1.2, která pojednávala o struktuře datového rámce vyskytujícího se na `MAC` vrstvě.

Ke každému datovému rámci je k dispozici 4bitový **řídící signál**, který slouží jako identifikátor kontrolních dat. Pozice logických `1` určují číslo datové linky, jež obsahuje kontrolní kódy, které byly zmíněny v tabulce 3.1. Nulové pozice tohoto signálu stanovují byty, ve kterých se nacházejí data. [27]

4 Popis verifikovaného návrhu

Cílem této práce je provést verifikaci obvodu, jehož blokové schéma je zobrazeno na obrázku 4.1. Jedná se o Ethernetové PHY, které bylo vytvořeno za účelem nízkolatenčního přenosu dat. Blokově je velice podobné klasickému PHY, ale některé jeho součásti byly uzpůsobeny tak, aby mohly být k takovéto komunikaci použity. Na jeho vysílací straně se objevuje proprietární LII rozhraní, jehož funkcionalita je popsána v sekci 5.1. Dále se zde nachází blok TX MAC, který v sobě obsahuje součásti jako CRC sloužící pro výpočet FCS, zpožďovací registry, jež obsahují přenášená data, a blok Preambule, kde je, jak název vypovídá vytvářena Ethernetová preambule. Výstupy již zmíněných částí jsou slučovány do 32bitového rámce pomocí multiplexoru nesoucího označení MX.

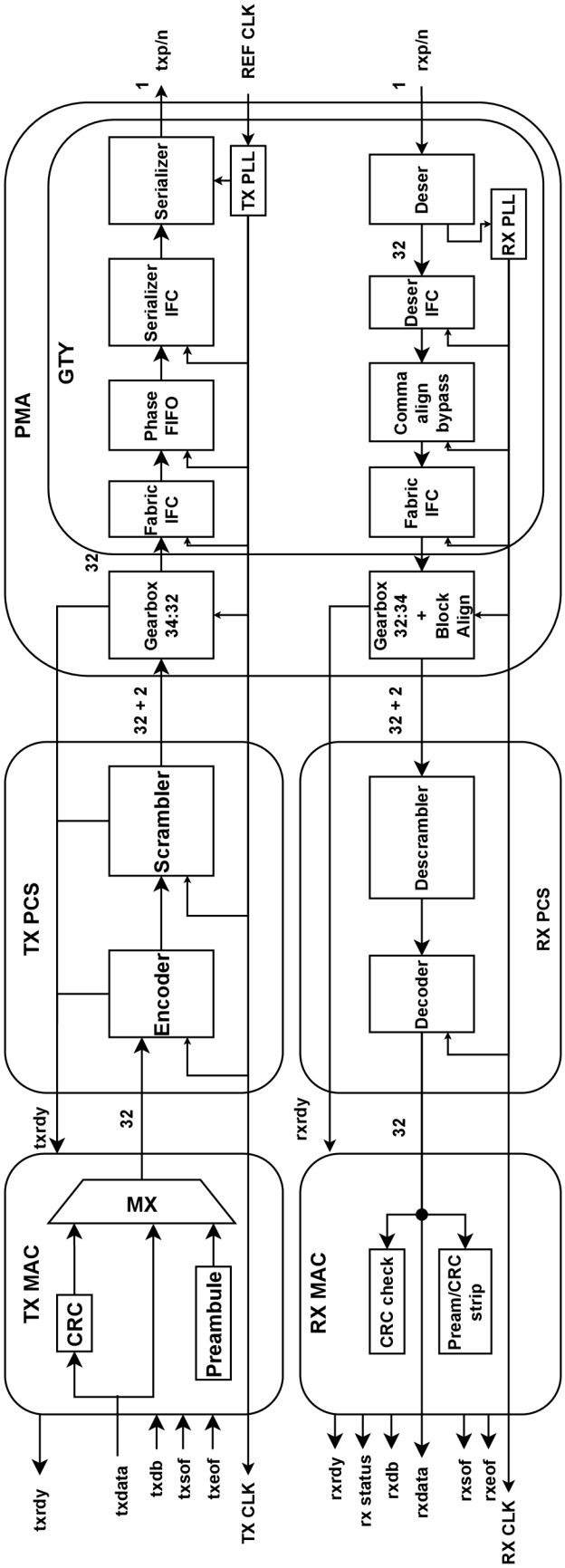
Dalším neméně důležitým blokem je TX PCS, uvnitř kterého se nachází Encoder, jenž má za úkol zakódovat příchozí rámce do příslušných sekvencí, k čemuž využívá tabulku 3.5, a přidělit jim patřičnou synchronizační hlavičku. Kvůli zrychlení a zjednodušení nepočítá se sekvencemi ordered set a je upraven tak, aby dokázal pracovat s již zmíněnou sběrnicí LII. Následuje Scrambler, ten je ve své podstatě implementován stejným způsobem, jak jej popisuje standard. Posledním článkem je opět zpožďovací registr, který není nativně používán, ale je ho možné zapnout nastavením parametru `SCR_FLOP_ENABLE` na hodnotu `true`. Výstupem tohoto bloku je 32bitový rámec obsahující 2bitovou hlavičku.

Poslední částí v TX směru je PMA. Tato komponenta obsahuje Gearbox, který slouží k zařazení hlavičky do datové části a odeslání takto upravených dat GTY transcieverům. Jelikož druhá část tohoto bloku není součástí návrhu, ale je vytvořena společností Xilinx, není ani součástí výsledné verifikace, tudíž ji není potřeba podrobněji popisovat. Jediná komponenta, jež bude pro testování důležitá, je serializér, který převádí přijatá data na tok bitů, výstupem tohoto bloku je tedy 1bitová sběrnice.

Nyní bude probrána RX strana Ethernetového PHY, na jejímž vstupu se nacházejí již zmíněná data serializéru. Ta jsou za pomoci deserializéru, který se nachází uvnitř GTY transceiverů, převedena opět na 32bitová slova a v kooperaci s Gearboxem rozdělena na datovou část a hlavičku. Jelikož přijatý rámec nemusí začínat hlavičkou, tak zde leží i synchronizační monitor, jenž je určen k nalezení začátku slova (hlavičky), čímž je Gearboxu umožněno správné vykonávání své funkce. Výstupem je tedy opět 32bitové datové slovo, 2bitová řídící hlavička a signál `rdy`, který říká, že jsou data validní.

Následuje blok RX PCS, kde jsou nejprve descramblována přijatá data, případně zpožděná v příslušném registru, a odeslána k dekódování dekodéru, jenž plní inverzní funkci, jako již zmíněný kodér. Poslední částí je RX MAC, který má za úkol u přija-

tých dat osekávat a kontrolovat CRC a preambuli. K tomuto účelu jsou definovány bloky CRC check a Pream/CRC strip. Opět jsou zde provedeny úpravy umožňující komunikaci se sběrnicí LII.



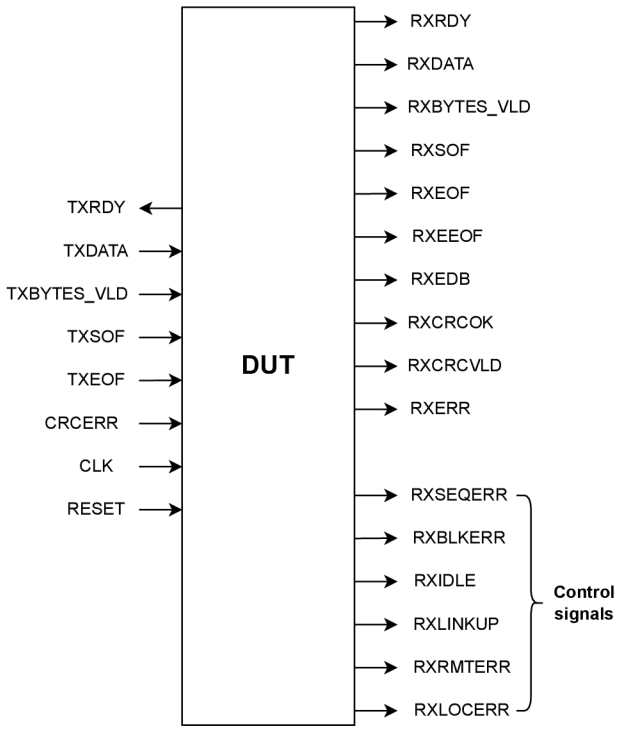
Obr. 4.1: Blokové schéma ethernetového PHY

5 Návrh verifikačního prostředí

V této kapitole bude podrobně popsáno vytváření verifikačního prostředí pro návrh nízkolatenčního ethernetového PHY za pomoci metodiky UVM a programovacího jazyka System Verilog. Pro potřeby verifikace zmíněného DUT a jeho dílčích komponent byly vytvořeny dvě prostředí, jedno pro vstup a druhé pro výstup. Z počátku této kapitoly bude detailně popsáno rozhraní LII, zejména jeho funkcionalita a návrh. Následně bude objasněn popis návrhu jednotlivých prostředí a komponent, jež se v nich nachází.

5.1 Rozhraní LII

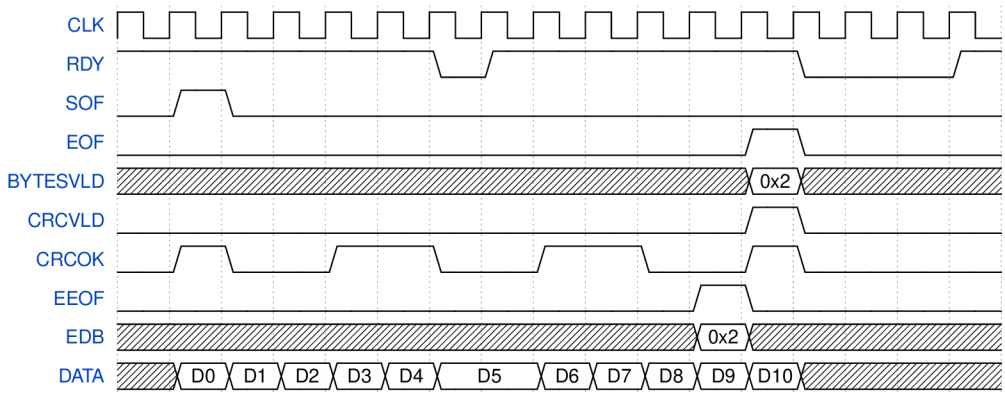
Pro účely verifikace bylo vytvořeno verifikační rozhraní LII nacházející se v souboru `interface.sv`. Struktura je zobrazena na obr. Jedná se o proprietární rozhraní vytvořené tak, aby jej bylo možné použít k implementaci nízkolatenčního Ethernetového PHY. 5.1. Je specifikováno jako náhrada za `XGMII` sběrnici, jejíž cho-



Obr. 5.1: Struktura LII sběrnice [27]

vání bylo popsáno v sekci 3.3. Jeho chování lze připodobnit k protokolu `FrameLink` s tím, že klientovi není umožněno zastavit odesílání dat. Z vývojářského pohledu to znamená, že neexistuje signál `source ready` signalizující připravenost v TX směru

a `destination ready` v RX směru. Z výše uvedeného vyplývá, že aplikace musí být schopná zpracovávat data při plné přenosové rychlosti. Na druhou stranu ethernetové PHY má umožněno vkládat čekací cykly, které jsou signalizovány porty `RXRDY` na přijímací straně a `TXRDY` na vysílací straně, jež se ve své podstatě chovají jako signál `clock enable` pro datové pipeline. Aby bylo umožněno ethernetovému PHY vynucování mezipaketových mezer, je tento signál na konci slova nastaven do logické 0 na dobu definovanou standardem (9,6 ns). Dále sběrnice obsahuje signály `RXSOF` a `TXSOF`, které označují začátek rámce.



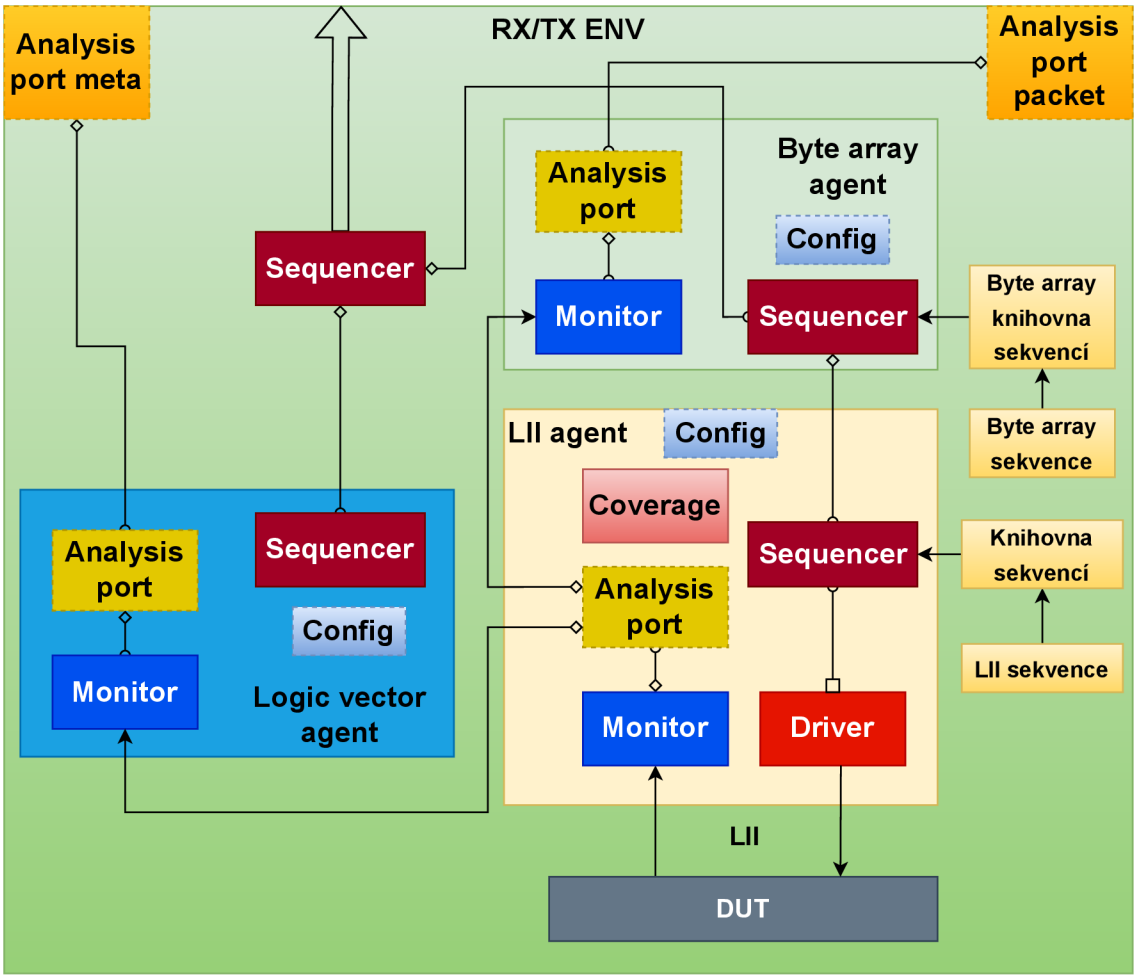
Obr. 5.2: Časový diagram komunikace přes LII

Další funkcionalitou, kterou toto rozhraní poskytuje, je přenos nezarovnaných dat. Tato skutečnost je určena počtem validních bytů a definují ji porty `RXBYTES_VLD` a `TXBYTES_VLD`. Tato hodnota je validní pouze na konci slova, který je na tomto rozhraní signalizován nastavením signálu `RXEOF` na přijímací straně a `TXEOF` na vysílací straně do logické 1. Povolené hodnoty signálů `RXBYTES_VLD` a `TXBYTES_VLD` jsou `0x0 - 0x4` hexadecimálně. Ostatní hodnoty jsou zakázané a mohou vést k neočekávanému chování GBASE-R kodéru. Nulový počet validních bytů značí ukončení rámce bez dat. Posledními definovanými porty jsou `EEOF` a `EDB`, které jsou posílány současně a říkají, že následující hodinový takt bude nastaven `EOF` do logické 1 s počtem validních bytů odpovídajících hodnotě `EDB`. Tyto dva signály jsou použity, aby RX MAC mohl kontrolovat hodnotu CRC paralelně se zpracováváním datového rámce. Poslední funkcionalitou, která je pro základní rozhraní definována je parametr `FAST_SOF`, jenž říká, zda-li se používá v dané verifikaci signál `EEOF`. Na TX straně je navíc definován signál `CRCERR`, kterým je úmyslně poškozována hodnota CRC. Tato funkce je využita k testovacím účelům uživatele. Na RX straně jsou dále navíc definovány porty `RXCRCOK` a `RXCRCVLD`, které slouží k signalizaci výsledku kontroly CRC. Následně je zde možné vidět port `RXERR`, jenž je použit k signalizování chyb v rámci, takto označená data jsou posléze vyšší vrstvou zahazována. V poslední

řadě jsou definovány kontrolní signály, definující o jaký chybový stav se jedná. Na obrázku 5.2 je znázorněna pomocí časového diagramu ukázková komunikace po této sběrnici, která je obdobná jak pro RX, tak pro TX směr.

5.2 Byte Array to LII prostředí

Prvním prostředím je Byte Array to LII (viz. obrázek 5.3), které se stará o konverzi Byte Array transakcí na LII transakce. Je rozdělené na tři části: RX (`rx_env_base`), jež má za úkol generovat a monitorovat nízkoúrovňové datové transakce, TX (`tx_env_base`), jehož úkolem je se starat o generování a monitorování `RDY` signálu, a RX ETH PHY (`rx_eth_phy_env_base`), které je využito specificky pro Ethernetové `PHY`. Funkcionalita všech tří prostředí se liší pouze v `run_phase` a použitých



Obr. 5.3: Byte Array to LII prostředí

komponentách. Tyto rozdíly budou definovány na závěr. Všechny tři prostředí mají

tedy za úkol vytvořit vrstvy Byte Array, LII a logic vector a vzájemně je propojit. Prostředí se tedy skládá z Byte Array agenta, LII agenta a logic vector agenta. Jelikož je nutné generovat data i metadata, je zde definován sekvencer, jenž sdružuje obě tyto varianty do jedné, kde je možné si vybrat, který z nich bude používán. Tento sekvencer je pak připojený na virtuální sekvencer v komponentě Testbench patřičného DUT, aby bylo možné jejich spouštění souběžně.

Agenti navíc disponují konfiguračním souborem, který definuje, zda-li je agent aktivní nebo pasivní. Jelikož LII agent komunikuje s rozhraním, jeho konfigurační soubor obsahuje navíc jméno rozhraní, s nimž bude propojen, a jeho pokrytí. Samotné prostředí taktéž obsahuje konfigurační soubor, který předává zmíněné parametry ostatním konfiguračním souborům. Navíc obsahuje proměnnou obsahující typ sekvence, jež udává pro jakou komponentu je sekvence určena.

Prostředí je rozčleněno do fází `build_phase`, `connect_phase` a `run_phase`, které definuje UVM metodologie.

`build_phase` je fáze, v níž jsou vytvořeny zmíněné komponenty, aby je bylo možné používat napříč verifikačním prostředím. Ke konfiguraci agentů je nutné získat konfigurační soubor prostředí. Jelikož není top komponentou a je potřeba docílit co největší znovupoužitelnosti, je k získání tohoto souboru použita UVM databáze (viz. 5.2, do které top komponenta uložila tuto konfiguraci prostředí.

```
if(!uvm_config_db #(config_item)::get(this, "", "m_config", m_config))
begin
    `uvm_fatal(get_type_name(), "Unable to get configuration object")
end
```

Ta je posléze přiřazena agentům a uložena do databáze, aby ji mohli používat dílčí komponenty agentů. Další důležitou částí je přetypování monitoru prostředí na Byte Array monitor (viz. 5.2), aby jej bylo možné propojit s Byte Array agentem. Poslední důležitou skutečností je v případě aktivního agenta vytvoření zmíněného sekvenceru.

```
byte_array::monitor::type_id::set_inst_override(
    monitor_byte_array #(DATA_WIDTH, \texttt{DIC}_EN, VERBOSITY,
                         META_WIDTH)::get_type(),
                         {this.get_full_name(),
                         ".m_byte_array_agent_rx.*"} );
```

V `connect_phase` je propojen LII agent s Byte Array a Logic Vector monitorem tohoto prostředí a funkčním pokrytím skrze Analysis porty (o monitoru více v sekci 5.2.5). Dále je zde nutné uložit do databáze zmíněný multifunkční sekvencer, aby

o něm měla povědomí nízkoúrovňová sekvence, která ho využívá k žádání o vysokoúrovňové transakce. Následně jsou patřičné sekvencery přiděleny daným částem uvnitř multifunkčního sekvenceru.

Poslední neméně důležitou fází je `run_phase`. Zde jsou definované použité sekvence. Tyto sekvence jsou podle typu komponenty uloženy do knihovny sekvencí, která je randomizována, aby bylo docíleno náhodného spouštění sekvencí. V této fázi je využit multifunkční sekvencer z předchozích fází a to tak, že je na něm spuštěna již zmíněná knihovna sekvencí. Jsou zde definovány čtyři sady sekvencí, které je možné vybrat a přiřadit komponentě na základě typu sekvence, jež se nachází v konfiguračním souboru. Rozdílem mezi RX, RX ETH PHY a TX `run_phase` jsou použité sekvence a již zmíněné typy agentů, kde každý z nich používá jiný typ rozhraní. Pro první dva případy RX prostředí jsou použity sekvence generující data a pro TX prostředí sekvence generující signál `RDY`.

5.2.1 Byte Array agent

Jedná se o vysokoúrovňového agenta, jehož úkolem je generovat pole bytů, které je využívané jako datová část nízkoúrovňových sekvencí. Skládá se z monitoru a sekvenceru, jež se starají o generování a monitorování vysokoúrovňových datových sekvencí, a Analysis portu. Agent je opět na základě UVM metodologie rozdělen do fází `build_phase` a `connect_phase`. V první fázi je opět použita databáze k získání konfigurace agenta, ze které je zjištěno, jestli je aktivní nebo pasivní. V prvním případě je vytvořen monitor a sekvencer. V opačném případě je vytvořen pouze monitor. Ve druhé fázi je propojen agent s příslušným monitorem skrze zmíněný Analysis port.

Sequence item

Byte Array sequence item, jinak též transakce, je využívána jako objekt pro přenos dat ostatním verifikačním komponentám. Ty ji mohou různě upravovat a využívat metody, které poskytuje. Tato transakce obsahuje pole data, do nějž je možné ukládat datové byty. Disponuje metodou pro kopírování transakce `do_copy`, porovnávání `do_compare` a tisk do konzole `convert2string`.

Monitor

Byte Array monitor slouží pouze jako rodičovská třída pro monitor prostředí Byte Array to LII. Funkcionálně pouze vytváří Analysis port a sequence item, které jsou využívány zmíněným monitorem.

Název	Popis
sequence_simple	Jednoduchá sekvence, jejímž účelem je generovat datové rámce o náhodné délce.
sequence_simple_const	Sekvence, která slouží ke generování rámců o konstantní délce.
sequence_simple_gauss	Sekvence generující rámce, jejichž délka je náhodná podle Gaussova rozdělení.
sequence_simple_inc	Sekvence, která vytváří rámce, jejichž délka je inkrementována do dané meze s příslušným krokem.
sequence_simple_dec	Sekvence, která vytváří rámce, jejichž délka je dekrementována po danou mez s příslušným krokem.
sequence_simple_meas	Sekvence, jež slouží k měřícím účelům. Generuje rámce, jejichž délka je inkrementována od 64 B do 1500 B. Byla využita k měření propustnosti a latence.

Tab. 5.1: Tabulka Byte Array sekvencí

Sekvencer a sekvence

Sekvencer na této úrovni je jednoduchý a dědí pouze metody a parametry od třídy `uvm_sequencer`. Sekvence této vrstvy má za úkol stanovit, jakým způsobem bude vypadat zmíněný `sequence_item`. V tomto případě definuje obsah pole data a počet vygenerovaných sekvencí, který je omezen rozsahem za pomoci `constraint c1`. Generování sekvencí probíhá v části body, která používá makro `‘uvm_do_with`, jež má za úkol provést první 4 kroky nutné k vytvoření sekvence (viz. 2.1.3). Rozsah možných velikostí pole je stanoven parametrem `inside` a proměnnými `data_size_min` a `data_size_max`. Takovýchto sekvencí může být větší množství a mohou definovat například různé typy rámců, od nejmenších rámců po Jumbo rámce.

Typy sekvencí

Pro účely otestování chování verifikovaných komponent byly vytvořeny různé typy Byte Array sekvencí, které generují odlišnými způsoby rámce, zejména upravují jejich délku. Výčet a popis těchto sekvencí je zobrazen v tabulce 5.1.

5.2.2 LII agent

Nízkoúrovňový agent navržený pro rozhraní LII, ve kterém jsou definovány komponenty, jež se starají o generování a monitorování nízkoúrovňových LII transakcí,

je oproti předchozímu vysokoúrovňovému agentovi rozdělen na RX a TX stranu, kvůli již zmíněnému `RDY` signálu, a navíc obsahuje driver, pomocí kterého je zapisováno na LII sběrnici. V poslední řadě bylo potřeba speciálně implementovat agenta RX_ETH_PHY pro výstup Ethernetového PHY, jenž se svým chováním lišil od ostatních, zejména chováním monitoru a některých výstupních signálů. Agenti jsou opět rozděleny na `build_phase` a `connect_phase`. V první fázi, stejně jako v předešlých případech, je vytažena konfigurace, na základě níž je zjištěno, o jaký typ agenta se jedná, a na základě toho jsou vytvořeny jeho komponenty. Oproti Byte Array agentovi je zde navíc vytvořen již zmíněný driver. Ve druhé fázi je použita databáze pro zjištění rozhraní, se kterým má monitor a driver kooperovat. Dále je zde vytvořeno propojení agenta s monitorem a driverem (pokud je aktivní) skrze Analysis port. Na závěr je `seq_item_export` LII driveru připojen na `seq_item_export` sekvenceru.

Sequence item

LII transakce je objekt, který definuje jednotlivé signály rozhraní pomocí proměnných. Následně je pak používaná ve verifikačním prostředí ke komunikaci mezi komponentami. Podobně jako Byte Array transakce obsahuje metody pro kopírování, porovnávání a tisk do konzole.

Driver

LII driver má za úkol převádět transakce na signály LII rozhraní. Opět je kvůli `RDY` signálu rozdělen na RX, RX ETH PHY a TX část. Na základě fázovacího systému UVM definuje `run_phase` všech tří typů driveru. První z nich má následující funkcionalitu. Nejprve pomocí metody `try_next_item` zkusí, jestli je dostupná nějaká sekvence. Pokud je dostupná (proměnná `req` neobsahuje hodnotu `null`), je převedena na signály a zapsána na rozhraní. K tomu jsou využity již zmíněné `clocking` bloky. Po tomto procesu je vytvořena odpověď obsahující hodnotu `RDY` signálu, která je odeslána sekvenci spolu s ID žádosti. Důvod odesílání odpovědi bude odůvodněn v sekci 5.2.4. Na závěr zkontrolujeme, zda vygenerovaná sekvence nemá nastavenou položku link status v logické 0. Pokud je tomu tak, počká se 125 us a sesynchronizuje se na náběžnou hranu. Pokud sekvence není dostupná (proměnná `req` obsahuje hodnotu `null`), jsou na rozhraní zapsány nulové hodnoty signálů `SOF` a `EOF`.

TX driver má za úkol zapisovat na rozhraní transakce obsahující `RDY` signál protistrany. Funkcionalita je podobná RX straně. Opět je nejprve požádáno o sekvenci, pokud existuje, transakce je převedena na signál a zapsána na rozhraní. V tomto případě není nutné odesílat odpověď, protože ji TX sekvence nevyužívá. Pokud žádost neexistuje, `RDY` signál je nastaven do logické 0.

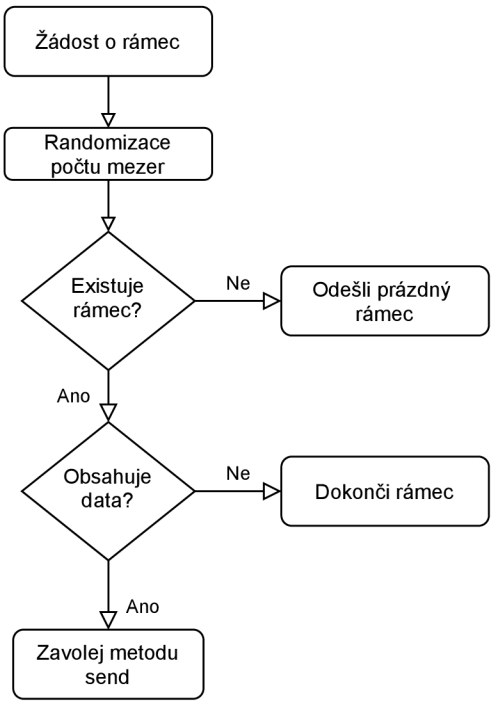
V poslední řadě se zde nachází RX ETH PHY driver, který je specifický pro vstup Ethernetového PHY. Chováním je podobný původnímu RX driveru s tím, že se zde nekontroluje signál link status, jelikož není jeho vstupem, ale výstupem, a komponenta si tedy tuto logiku řeší sama.

Monitor

Jak již bylo zmíněno, LII monitor je komponenta, která má za úkol vzorkovat signály patřičného rozhraní a konvertovat je na položky dané transakce. K tomuto účelu využívá tzv. virtuální rozhraní, jež kopíruje signály toho reálného. Podle UVM metodologie je rozdělen na `build_phase` a `run_phase`. V první fázi je zkonstruována transakce a Analysis port. Následuje již zmíněná transformace signálů na transakční položky a zápis na Analysis port.

5.2.3 RX Sekvence

Pro účely generování transakcí pro vstupní rozhraní verifikované komponenty byla zkonstruována knihovna, obsahující různé druhy RX sekvencí. V této sekci bude popsána jejich struktura. Tato sekvence má za úkol definovat, jakým způsobem budou vypadat data nízkoúrovňových transakcí. Funkcionalita, která je pro většinu sekvencí stejná, je popsána vývojovým diagramem 5.4.



Obr. 5.4: Vývojový diagram funkce LII sekvence

Název	Popis
sequence_simple_eth_phy	Sekvence generující bezchybné LII transakce pro Ethernetové PHY
sequence_simple_eth_phy_ const_gaps	Sekvence genrující LII transakce pro Ethernetové PHY s konstaními mezerami.
sequence_simple_eth_phy_ err_crc	Sekvence generující LII tranaskce pro Ethernetové PHY, které obsahují špatnou hodnotu CRC.
sequence_simple_eth_phy_ no_gaps	Sekvence generující LII transakce pro Ethernetové PHY s plnou propustností.
sequence_simple_rx_ random_errors	Sekvence generující LII transakce pro RX MAC s náhodně se vyskytujícími errory
sequence_simple_rx_ random_link_status	Sekvence generující LII transakce pro RX MAC, kde je náhodně shazována linka.
sequence_simple_rx_error_ with_random_link_status	Sekvence generující LII transakce pro RX MAC, kde je náhodně shazována linka v kombinaci s vyskytujícími se errory.
sequence_simple_rx_ random_sof	Sekvence generující LII transakce pro RX MAC, jejichž SOF je umístěn v náhodné transakci.
sequence_simple_rx_ sof_after_eof	Sekvence generující LII transakce pro RX MAC, jejichž SOF je umístěn v transakci za poslední transakcí rámce.

Tab. 5.2: Tabulka LII sekvencí

V prvním kroku je za pomoci UVM databáze získán vysokoúrovňový sekvencer, bez kterého by nebylo možné generovat datové rámce a metadata pro tuto sekvenci. Následuje samotné získání vysokoúrovňové sekvence a logic vector sekvence pomocí metody `try_get`, jež se volá cyklicky, dokud není vytažena poslední sekvence. Dále je zjištěno, jestli je již zmíněná Byte Array sekvence vytvořena. Pokud ne, je vygenerována prázdná sekvence obsahující nulové hodnoty veškerých položek transakce. V opačném případě je dále zkontrolováno, zda-li obsahuje data. V negativní případě je odesláno sekvenceru potvrzení o zpracování sekvence pomocí `item_done`. Jestliže rámec obsahuje data, je zavolána metoda `send`, která se již liší na základě zvolené sekvence. Pro dosažení co největšího pokrytí bylo vytvořeno 9 typů sekvencí. Aby bylo umožněno jejich průběžné spouštění, byly umístěny do sekvenční knihovny. Funkcionalita každé z nich je popsána v tabulce 5.2.

Každá z těchto sekvencí má z úkol vytvářet transakce, které mají definovaný formát dle standardu. Nejprve je vygenerován náhodný počet mezer (výjimkou je

sekvence a pevným počtem mezer a bez mezer). Jakmile je tato operace dokončena, je dle specifikace MAC vrstvy poslána první transakce obsahující `SFD` identifikátor spolu s nastaveným `SOF` v logické 1, který je následován datovým slovem obsahujícím preambuli. Po vygenerování těchto počátečních identifikátorů je započato odesílání dat. Po každé vytvořené transakci je po dokončení zkontrolována hodnota signálu `RDY`. Pokud je během procedury odesílání dat zpozorováno, že po současné transakci následuje poslední, je u předposlední transakce nastaven signál `EEOF` do logické 1 a jsou vypočítány validní byty posledního datového slova, které jsou uloženy do signálu `EDB`. Následující transakce je, jak již bylo zmíněno, poslední, a proto je nastaven `EOF` a příslušný počet bytů v položce `BYTES_VLD`. Na závěr dojde k jejímu odeslání. Formát různých sekvencí se pak liší v nastavení příslušných bitů, jež vyvolají v DUT patřičnou situaci, kterou vysvětluje políčko popis u každé sekvence.

5.2.4 TX Sekvence

Pro účely generování signálu `RDY` byla vytvořena knihovna sekvencí, jež obsahuje dvě sekvence, které se starají o jeho generování buď náhodně, nebo jednou za 32 taktů. Jejich funkcionalita je velice jednoduchá. V nekonečné smyčce jsou vyplňovány žádosti příslušnou hodnotou `RDY`. Takováto položka je následně odeslána a ukončena.

5.2.5 Byte Array to LII monitor

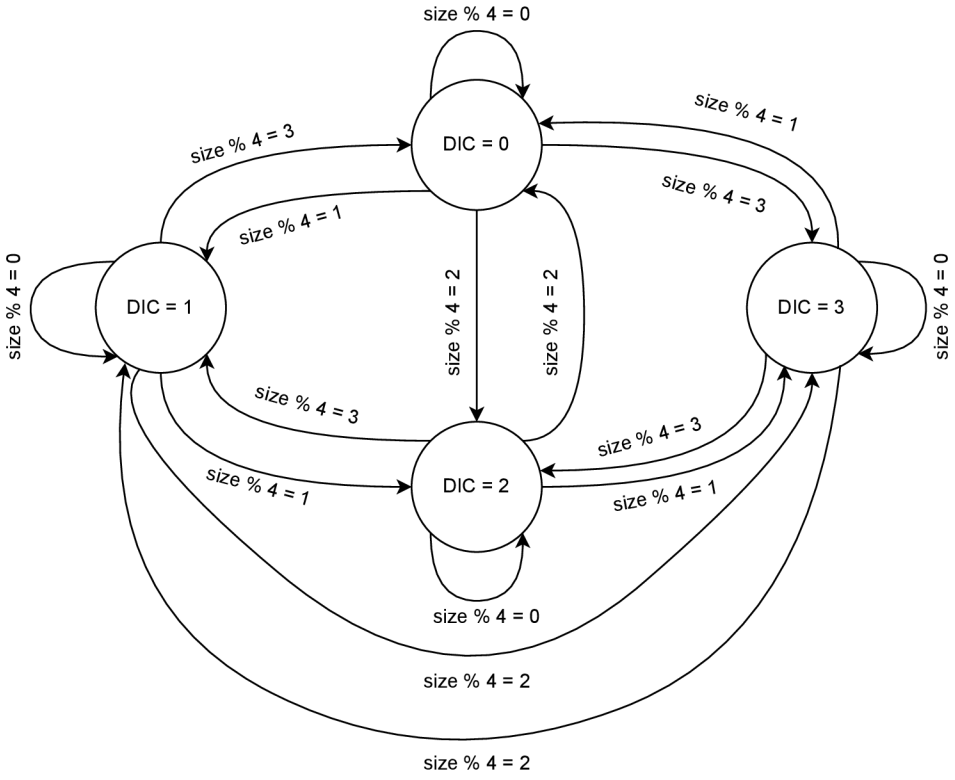
Tento monitor má za úkol přijímat nízkoúrovňové transakce a na základě jejich obsahu je skládat do vysokoúrovňové transakce. Byl vytvořen, aby je bylo možné odesílat komponentě Scoreboard ke kontrole. Skládá se z Byte Array monitoru, který implementuje metodu write, jejímž účelem je Sestavovat Byte Array sekvence a provádět kontrolu mezirámcových mezer. Druhým velkým celkem je Logic Vector monitor, jenž na základě vlastností LII transakcí ukládá stavová data do Logic Vector transakcí.

Kontrola mezirámcových mezer

Jelikož na této vrstvě mohou být mezi rámce vkládány mezery, které musí dosahovat patřičných hodnot, je nutné provádět jejich kontrolu. K tomuto procesu je využíván algoritmus Deficit Idle Count (`DIC`), jehož funkcionalita je popsána stavovým automatem na obrázku 5.5, jenž generuje referenční hodnotu mezer.

Princip tohoto algoritmu využívá dvě klíčové proměnné. První z nich je tzv. Deficit Idle Count, který udává, kolik bytů mezer zbývá, aby byla dodržena minimální mezera. Druhou je počet nevalidních bytů na konci rámce, které jsou brány také jako

mezera a přičítají se k dosavadní hodnotě. Počátečním stavem algoritmu je nastavení hodnoty `DIC` do 0. V této chvíli jsou nejprve zkontrolovány poslední nevalidní byty rámce, jejichž počet je nastaven jako následující hodnota `DIC`, a je ustavena minimální hodnota referenční mezery. S příchodem dalšího rámce se situace opakuje s novým `DIC` s tím, že následující hodnota se pak dopočítává jako `DIC` + počet nevalidní bytů slova. Stejným způsobem je zacházeno s ostatními hodnotami `DIC` až do hodnoty `DIC` = 3. Tímto způsobem je docíleno toho, že rámce mezi sebou budou mít minimálně takové mezery, aby bylo možné udržet synchronizaci mezi vysílačem a přijímačem. Na závěr algoritmu je vždy přičtena vypočtená hodnota mezery k předchozím, aby bylo možné posléze vypočítat průměrnou mezeru.



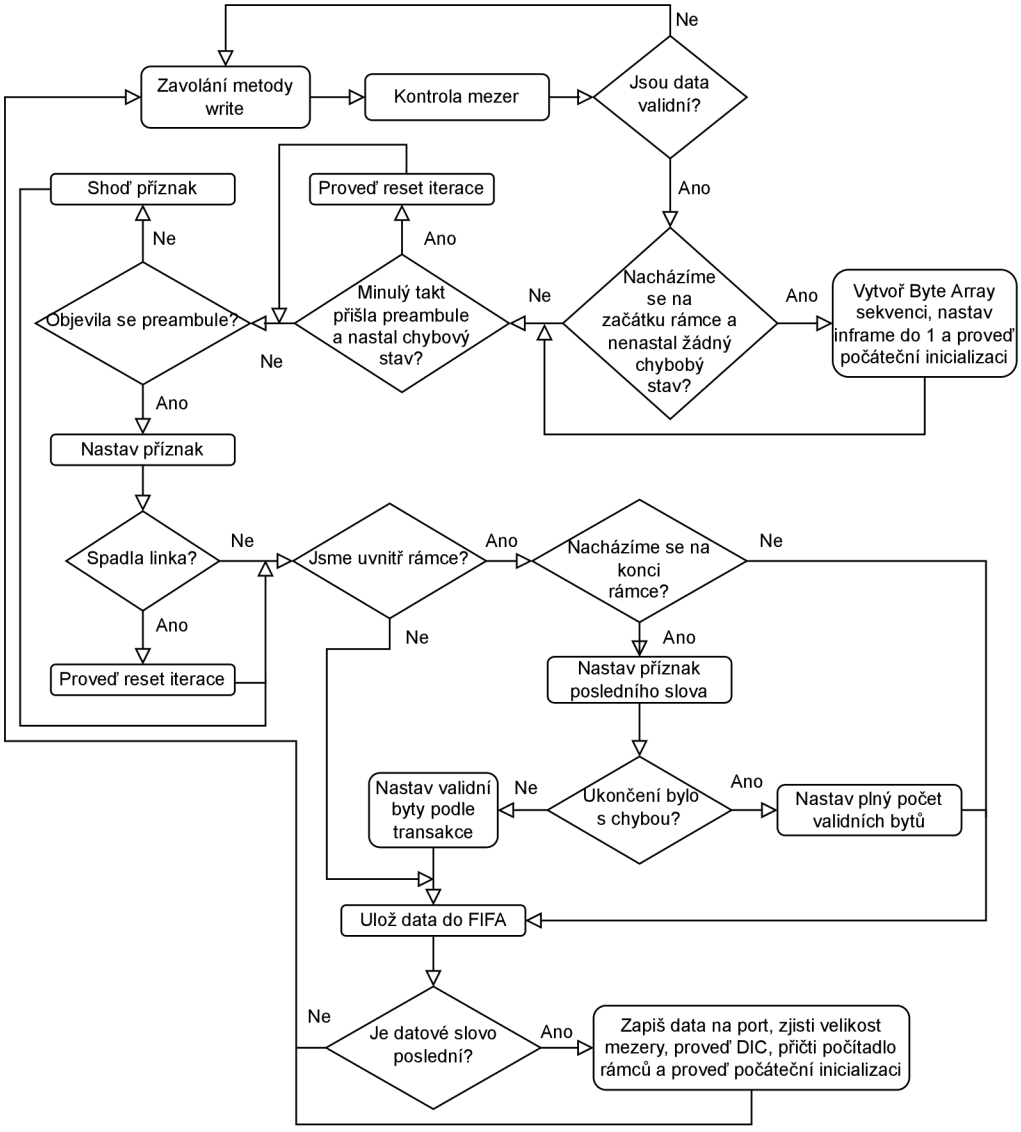
Obr. 5.5: Algoritmus Deficit Idle Count

Referenční mezery jsou využity pro porovnávání s reálnou hodnotou mezery, která je zajišťována metodou `gap_control`. Předtím, než je vykonána, se nejprve zjistí, kolik posledních bytů daného rámce je nevalidních a uloží se do čítače mezer. Pokud je příchozí transakce validní, je dále ověřeno, zda-li se nejedná o první rámec a transakci obsahující identifikátor jeho začátku. Pokud je tomu tak, proběhne přičtení poslední mezery a její validace. V případě, že se jedná o 200 rámec, je proveden i výpočet průměrné referenční a reálné mezery a její ověření. Pokud se neobjevil začátek rámce, probíhá inkrementace příchozích mezer. Tento proces se opakuje pro

každý příchozí rámec.

Sestrojování Byte Array sekvencí

Jak již bylo uvedeno na začátku kapitoly, monitor disponuje metodou `write`. Její princip je popsán vývojovým diagram na obrázku 5.6.



Obr. 5.6: Vývojový diagram metody write Byte Array to LII monitoru

Pokud je zavolána, dojde k provedení kontroly mezer za pomoci metody s názvem `gap_control` a k následné kontrole validity dat, která je jednoduše provedena ověřením signálu `RDY`. Pokud nejsou validní, dojde k opětovnému zavolání této metody. V opačném případě je zjištěno, zda-li transakce obsahuje korektní indikátor

začátku rámce `SOF`. Korektní ve smyslu, že během jeho přenosu nedošlo k chybovému stavu, tj. pádu linky nebo chybě způsobené přijetím transakce, která obsahuje nevalidní sekvenční indikátor tzv. sequence error. Se začátkem rámce je tedy posléze vytvořena byte array transakce a je nastaven příznak `inframe`, jenž říká, že se algoritmus nachází uvnitř rámce. Následně je promazána datová FIFO paměť a hodnota validních bytů `BYTE_NUM` je nastavena na počáteční hodnotu.

Dalším krokem algoritmu je kontrola stavu, kdy minulý hodinový takt byla v datech transakce obsažena preambule a následně nastal jeden z již zmíněných chybových stavů. V takové situaci je ukončeno zaznamenávání rámce vynulováním příznaku `inframe` a je provedeno počáteční nastavení, tj. vynulování patřičných proměnných.

Následuje zaznamenávání stavu, kdy se v datech transakce objevila preambule. Zde je nastaven příznak `was_preambule`, který je využíván v předchozím kroku. Následně je ověřeno, jestli v tomto čase nedošlo k pádu linky. V takovém případě dojde opět k ukončení záznamu transakcí a počátečnímu natavení.

Dalším krokem je kontrola, zda-li se nacházíme uvnitř rámce. Pokud ano, provede se kontrola, jestli již nastal validní konec rámce označený signálem `EOF`. Korektní zakončení je signalizováno nastavením počtu posledních bytů, které se nachází v položce `BYTES_VLD`. Chybné zakončení, tj. zakončení jedním ze zmíněných chybných stavů, způsobí nastavení všech posledních bytů jako validních. V obou zmíněných případech dojde k nastavení příznaku `last_chunk`, který říká, že rámec má být zapsán na analysis port.

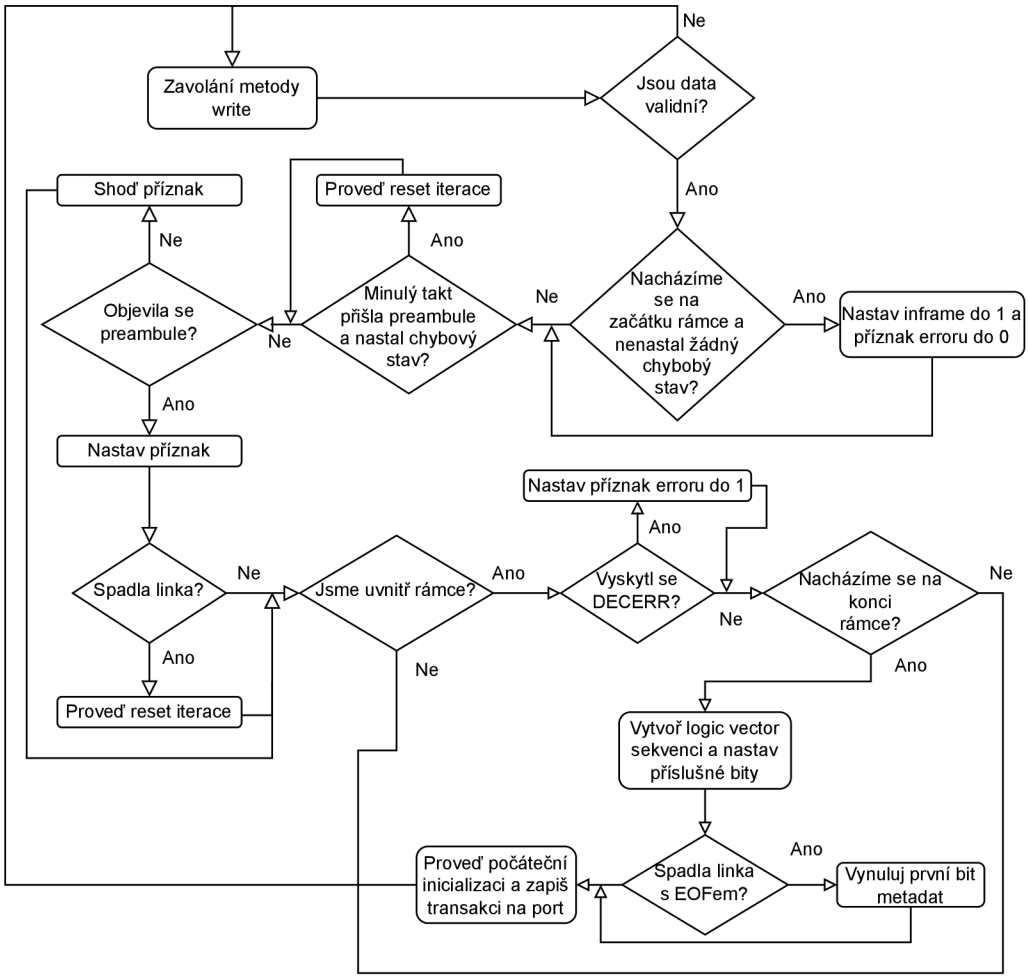
Posledním krokem je tedy vyčtení dat z FIFO paměti do byte array transakce na základě `last_chunk`, její zapsání na analysis port, provedení počátečního nastavení a inkrementování čítače rámců. V průběhu již zmíněných stavů dochází v případě validních dat k jejich zápisu do FIFO paměti. Veškeré zmíněné situace jsou kontrolovány s každým zavoláním metody `write`.

Sestrojování Logic vector sekvencí

Jelikož je potřeba některé části transakce dostat do scoreboardu, bylo nutné vytvořit Logic Vector Monitor, který tuto funkci vykonává. Jak je patrné z vývojového diagramu metody write 5.7, je principiálně podobný předchozímu, jelikož pro něj platí obdobná pravidla.

Celý proces začíná s příchodem validní LII transakce. Po nalezení bezchybného začátku rámce je provedena počáteční inicializace proměnných a příznakem `inframe` je oznámeno, že se algoritmus nachází uvnitř rámce. Následuje proces kontroly preambule, který byl popsán v předchozí sekci. Posléze je ověřeno, zda-li předchozí procesy nezměnili hodnotu `inframe`. Pokud je tomu tak, předchozí série kontrol je

provedena pro další rámec. V opačném případě, kdy se algoritmus stále nachází uvnitř rámce, započne samotné zaznamenávání chybových příznaků. Prvním z nich je již zmíněný `RXDECERR`, jehož výskyt zapříčiní nastavení příznaku `err_trig` do logické 1. Poslední fází je validní, či nevalidní ukončení rámce, které je iniciováno buď nalezením položky `EOF` v přijaté transakci, pádem linky nebo přijetím chybné sekvence. Chybné stavy jsou promítnuty v položkách `LINK_STATUS` a `RXSEQERR`. V těchto případech je vytvořena Logic Vector transakce a je naplněna příslušnými daty, jmenovitě je to příznak `err_trig` a položky `LINK_STATUS`, která je nastavena do logické 1, pouze pokud došlo k pádu linky a zároveň nedošlo i k validnímu ukončení rámce, `RXDECERR`, `RXSEQERR` a `CRCERR`. Na závěr je provedena počáteční inicializace proměnných a zápis na analysis port. Všechny tyto kroky jsou prováděny pro každou příchozí validní transakci.



Obr. 5.7: Vývojový diagram metody write Logic Vector monitoru

5.3 Byte Array to LII RX prostředí

Druhým typem prostředí je Byte Array to LII RX, které má obdobnou funkci jako předchozí prostředí, ale liší se strukturou. Z pohledu struktury se liší tím, že se skládá pouze z jedné části, a to `env_base`, protože se používá na RX straně a není potřeba pomocí něj cokoliv generovat. Obsahuje tak jako předchozí prostředí tři agenty. Byte Array a Logic Vector agenti jsou definováni stejně, proto nebudou v této sekci popisováni. Poslední LII agent se skládá pouze z monitoru a analysis portu, protože se od něj neočekává generování dat. Jako rozhraní je v tomto případě přiřazeno virtuální rozhraní `lii_if_rx`, o němž bude zmínka v sekci 5.4. Poslední důležitou komponentou, která se svou funkcionalitou liší od prvního prostředí jsou monitory, jejichž funkce bude podrobně popsána v sekci 5.3.1. Toto prostředí je aplikováno na RX straně Ethernetového PHY.

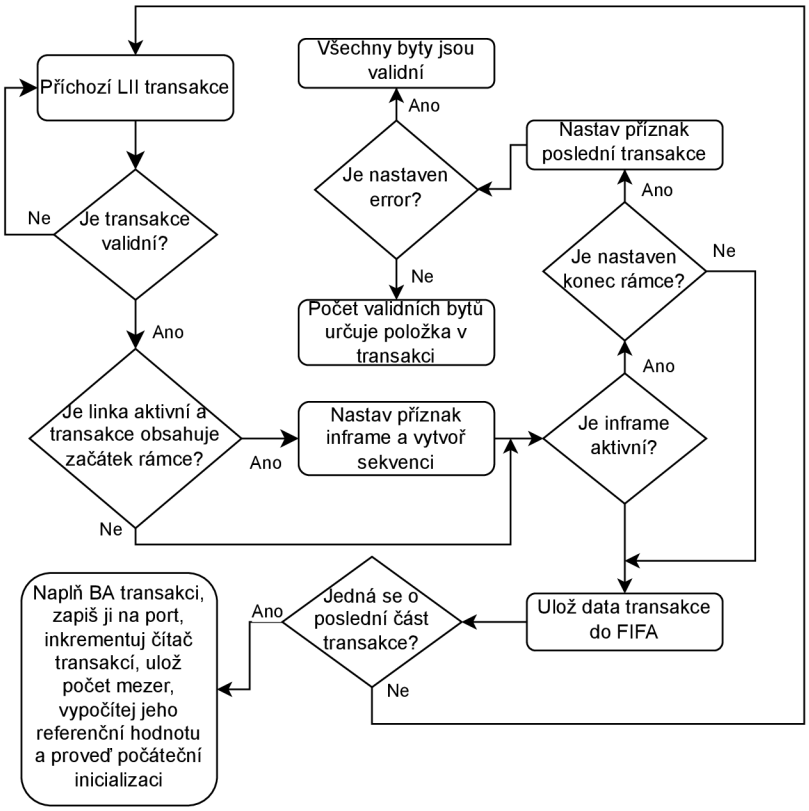
5.3.1 Byte Array to LII RX monitor

Slouží, tak jako předchozí přechodový monitor, pro převod nízkoúrovňových transakcí na vysokoúrovňové, které mohou být pak následně efektivněji porovnávány ve scoreboardu. Skládá se, tak jako předchozí monitor, ze dvou velkých celků. Monitoru pro převod LII transakcí na Byte Array transakce a monitoru sloužícímu k zapisování metadat na základě údajů v položkách nízkoúrovňových transakcí. Informace o metadatech jsou potřebné pro konstrukci modelu a porovnávání dat ve scoreboardu.

Sestrojování Byte Array sekvencí

První velký celek se skládá opět ze dvou částí a to logiky pro kontrolu mezer, která byla již popsána v kapitole 5.2.5 a logiky pro sestrojování Byte Array transakcí. Tuto funkcionalitu lze popsat následujícím vývojovým diagramem.

Z obrázku 5.8 je patrné, že celá iterace začíná přijetím LII transakce, ze které je zjištěno, jestli je validní na základě signálu `RDY`. Pokud tomu tak je, ověří se na základě signálu SOF, zda obsahuje začátek rámce, jenž je příznakem vytvoření Byte Array transakce a nastavení příznaku `inframe`, jenž říká, že se nacházíme uvnitř rámce. V takovém případě jsou průběžně ukládaná data do paměti typu FIFO a je kontrolováno, zda-li se neobjevila transakce obsahující položku `EOF` nastavenou do logické 1, nebo-li konec rámce. V takovém případě je nastaven příznak `last_chunk`, který říká, že tato data jsou poslední, a je zjištěno, zda taková transakce neobsahuje příznak `ERR`. Pokud jej obsahuje, jedná se o chybnou transakci a všechny byty dat transakce jsou uloženy do paměti. V opačném případě se použije položka `BYTES_VLD`



Obr. 5.8: Vývojový diagram logiky sestrojování Byte Array sekvencí

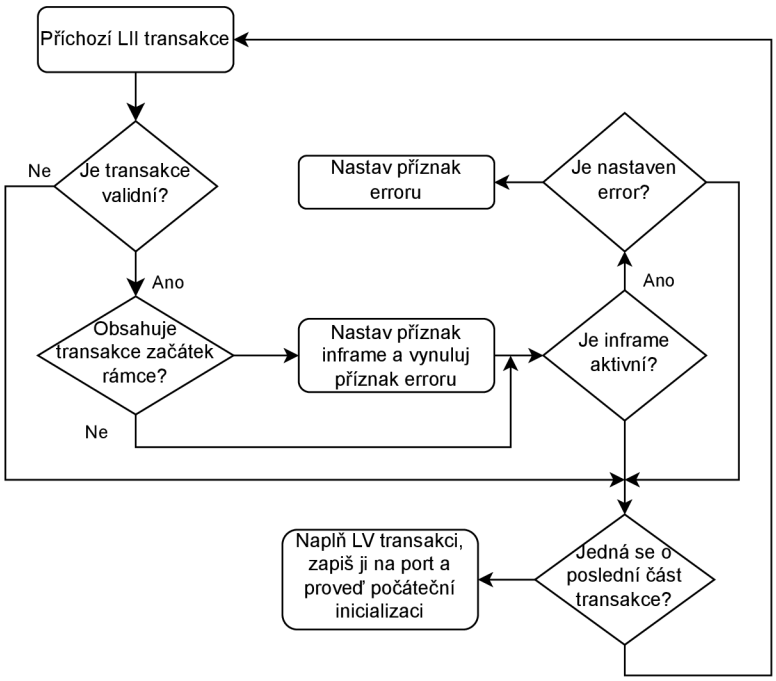
jako indikátor počtu validních bytů. V případě poslední transakce je následně naplněna Byte Array transakce daty z paměti a zapsána na Analysis port. Posléze je zjištěna hodnota mezirámcové mezery a vypočítána její reference. Obě tyto hodnoty jsou využívány v logice pro kontrolu mezer. Celý tento algoritmus je opakován pro každou přijatou LII transakci.

Sestrojování Logic vector sekvencí

Druhý velký celek slouží, jak již bylo zmíněno, pro zápis metadat do logic vector transakce. Jeho funkcionalita je popsána vývojovým diagramem na obrázku 5.9.

Celá iterace opět začíná přijetím transakce. Pokud je validní a jedná se o první transakci (`SOF` je v logické 1), je nastaven již zmíněný příznak `inframe` a je vynulován příznak erroru. Pokud se tedy algoritmus nachází uvnitř rámce, je kontrolováno, zda-li transakce obsahuje položku `ERR`, v takovém případě je nastaven příznak `err_trig`, kterým je řečeno, že někde uvnitř rámce se nachází chyba. V poslední části algoritmu je kontrolováno, jestli se jedná o poslední rámec a to na základě položky `CRC_VLD`. V takovém případě je logic vector transakce naplněna metadaty, jmenovitě se jedná o `err_trig` a položky `link_status`, `crc_ok` a `crc_vld`. Na závěr

je provedena počáteční inicializace (vynulování příznaků) a zápis na Analysis port. Celý algoritmus je opakován pro každou příchozí transakci.



Obr. 5.9: Vývojový diagram logiky sestrojování logic vector sekvencí

5.4 Verifikační rozhraní LII

Z důvodu verifikace Ethernetové fyzické vrstvy a jejích částí byla vytvořena dvě verifikační rozhraní `lii_if` a `lii_rx_if`. Jsou využívána k propojení verifikačního prostředí k DUT. První z nich téměř odpovídá vstupní části rozhraní LII, která byla zmíněna v sekci 5.1, ale přibyly signály `EEOF`, `LINK_STATUS`, `RXDECERR` a `RXSEQERR`, díky nimž je umožněno verifikovat komponentu RX PCS, tyto signály byly popsány již ve zmíněné kapitole pojednávající o LII rozhraní. V poslední řadě je zde signál `META` na přenos případných metadat. Tento typ rozhraní je používán v již zmíněném LII agentovi (viz. sekce 5.2.2).

Druhé rozhraní odpovídá reálnému rozhraní na výstupu Ethernetového PHY, které již bylo popsané dříve. Aby bylo možné připojit toto rozhraní k DUT, bylo taktéž potřeba vytvořit patřičného agenta, který se liší od LII agenta pouze tím, že nevyužívá sekvencer a je na něj aplikováno toto rozhraní. Přechodné prostředí z nižší LII úrovně na vyšší datovou Byte Array úroveň se liší pouze funkcionalitou monitoru.

Jak je patrné z předchozích informací, rozhraní jsou definována pro obě strany, z tohoto důvodu jsou pro porty použity datové typy `wire`, které to umožňují. Dále jsou definované tzv. `clocking blocks`, jež obsahují sadu signálů synchronizovaných na danou hodinovou doménu, jež slouží k obsluze driveru a monitoru. Jsou vytvořeny pro každý směr zvlášť. Dalším nástrojem jsou modporty `dut_rx`, `dut_rx_eth_phy` a `dut_tx`, které umožňují modulu DUT přistupovat k signálům tohoto rozhraní. Aby bylo možné zapisovat nebo číst data na/ze sběrnice, jsou definovány modporty `driver_tx`, `driver_rx`, `driver_rx_eth_phy` pro driver a monitor.

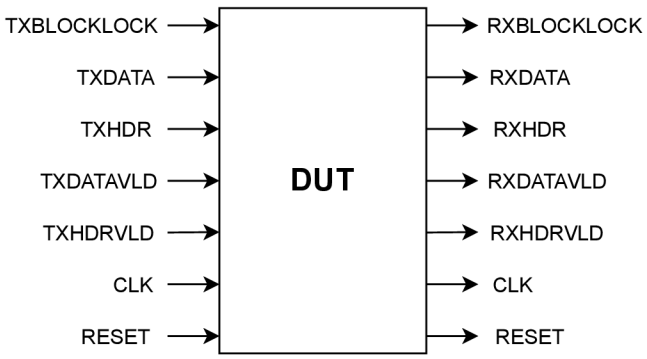
Pro obě rozhraní platí podmínky, jež vyplývají z popisu výše. Pro jejich ověření byl vytvořen modul `property.sv` za pomoci jazyka SVA. První podmínkou, která musí být splněna je, aby se po nastavení signálu `EEOF` do logické 1 nastavil signál `EOF`, toho je docíleno pomocí pravidla `eof_after_eeof`, kde je v případě nastaveného `RDY` v logické 1 s každým hodinovým taktem kontrolováno, zda-li se po nastavení `EEOF` do logické 1 objevil následující takt `EOF` v logické 1. Současně s tímto pravidlem je podobným způsobem prováděno porovnání signálu `EDB` s `BYTES_VLD` pomocí pravidla byte_vld_control. V posledním pravidle je ověřeno, jestli ke každému `SOF` existuje patřičný `EOF`, což znamená, že je rámec ukončen, tato problematika je ověřována pravidlem `sof_eof_control`. To v případě, že se objeví na sběrnici validní `SOF`, který spustí sekvenci `eof_seq`, která je ukončena až s příchodem signálu `EOF`. Pro případ rozhraní `LII_RX` je zde navíc kontrolováno, že s jakýmkoliv typem erroru je nastaven signál `ERR` do logické jedničky. Pokud není splněno alespoň jedno pravidlo z výše zmíněných, je vyhlášena patřičná chybová hláška. Kontrola těchto podmínek může být zapnuta pouze v případě, že nejsou úmyslně generovaná chybná data.

Další důležitou součástí rozhraní je soubor `coverage.sv`, který obsahuje informace o požadovaném funkčním pokrytí. Dle definice rozhraní byly definovány tzv. covergroups, v nichž je definován uzel, který se bude kontrolovat, a hodnoty, jež mají, případně nemají, nastat. První z nich je skupina `m_cov_rdy_sig`, která v sobě obsahuje uzel `rdy`. Tato skupina má zjišťovat, jestli je signál `RDY` různě dlouhou dobu v logické 1 a 0, v našem případě se buď střídá 1 s 0, nebo 1 trvá různě dlouhou dobu. Další skupinou je `m_cov_bytes_vld_sig`, která obsahuje uzel `bytes_vld`. Zde jsou definovány přípustné a nepřípustné hodnoty tohoto signálu, jež vyplývají z předešlých informací o sběrnici.

5.5 Rozhraní PMA

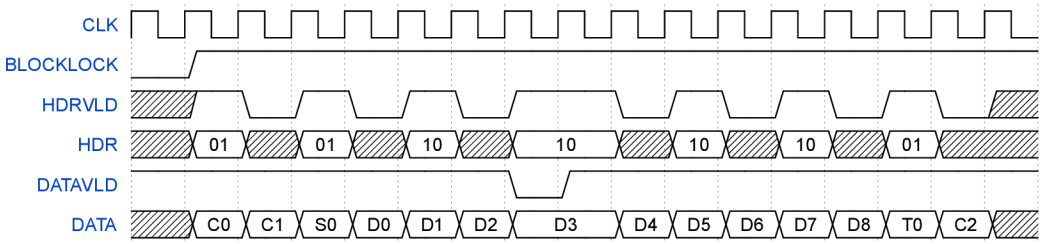
K účelu verifikování dílčích komponent ethernetového PHY bylo vytvořeno rozhraní PMA, které má popisovat vstupy a výstupy podvrstvy PCS. Tak jako LII se nachází v souboru `interface.sv`. Struktura této sběrnice je zobrazena na obrázku 5.10.

Popis jednotlivých vstupů a jejich funkcionalita vychází ze sekce 3.2.1, která popisuje podvrstvu PCS. Sestává z 5 portů, jež jsou stejné jak pro vstup, tak výstup.



Obr. 5.10: Struktura PMA sběrnice [27]

Prvním z nich je `BLOCKLOCK`, pokud je nastaven v logické 1, znamená to, že se podařilo najít hranici bloku. Tato skutečnost říká, že synchronizační monitor našel v datovém toku, který pochází z deserializéru a je zarovnaný gearboxem, první hlavičku, která se nachází na začátku slova. Pokud ji nenašel, je tento signál v logické 0 a nezapočalo vysílání dat kodéru. Dalším portem jsou `DATA`, jejichž obsahem jsou zakódovaná a scramblovaná 32bitová slova pocházející nebo vstupující do bloku PCS. Dále jsou zde signály `HDR` a `HDRVLD`. První z nich udává hlavičku, kterou vytváří kodér a jak bylo zmíněno v sekci 3.2.1, může nabývat pouze hodnot `01` a `10`, ostatní jsou nevalidní. Jelikož je sběrnice PMA 32bitová, musel být přidán právě signál `HDRVLD`, který říká, že se zrovna přenáší první z dvojic LII slov, a tudíž obsahuje validní hlavičku (`HDRVALID = 1`). V opačném případě se jedná o druhou část dat. Posledním portem je `DATAVLD`, který má stejnou funkcionalitu jako signál `RDY` nacházející se na LII sběrnici (v komponentě jsou přímo propojeni), tudíž je jeho funkcionalita patrná ze sekce 5.1. Ukázková komunikace po PMA sběrnici je patrná z obrázku 5.11.



Obr. 5.11: Časový diagram komunikace přes PMA

Na základě výše zmíněných informací byl vytvořen soubor `property.sv`, který ověřuje platnost pravidel komunikace po této sběrnici. Obsahuje zejména kontrolu

výskytu hlavičky v každém druhém slově, tj. souběžně se signálem `HDRVLD`, a jestli se na portu `HDR` nachází pouze platné hodnoty.

5.6 Byte Array to PMA prostředí

Byte Array to PMA je prostředí, které spravuje komponenty potřebné pro účely verifikace PCS podvrstvy. Jeho účelem je převádět vysokoúrovňové datové rámce na nízkoúrovňové transakce a naopak. Jelikož je velice podobný již zmíněnému LII agentovi, bude popsán stručněji. Obsahuje, tak jako předchozí, vysokoúrovňovou vrstvu ve formě Byte Array agenta sloužící ke generování dat, a nízkoúrovňovou, jež definuje chování PMA rozhraní. Nižší vrstvu řídí PMA agent, který se skládá z dvojice driver a monitor, jejichž účelem je oboustranná transformace PMA transakcí na jednotlivé signály PMA sběrnice. Dalšími důležitými komponentami je knihovna s patřičnými sekvencemi a sekvencer. Tato dvojice má za úkol generovat náhodný provoz pro testovaný návrh. V poslední řadě jsou zde konfigurační soubory sloužící k nastavení již zmíněných komponent. Jedinými odlišnostmi od předchozího prostředí jsou definované sekvence, nacházející se v knihovně, a monitor sloužící k transformaci vysokoúrovňových datových rámců na nízkoúrovňové transakce. Tyto komponenty budou popsány v dalších sekcích.

5.7 PMA sekvence

Za účelem generování transakcí pro vstupní rozhraní podvrstvy PCS byly vytvořeny PMA sekvence, jejichž účelem je definovat, jakým způsobem budou vypadat nízkoúrovňové PMA transakce. Před započetím generování sekvencí je nejprve vytažen z UVM databáze příslušný vysokoúrovňový sekvencer, následně je obdobným způsobem jako tomu bylo u LII sekvence provedeno vytažení vysokoúrovňové transakce a její kontrola. Toto ověřování platí pro všechny typy PMA sekvencí

Pokud jsou všechny podmínky splněny, započne generování příslušné sekvence. Datová slova vznikající v průběhu generování jsou, jak již bylo zmíněno 32bitová, proto jsou generována vždy po dvojicích (Kodér vytváří 64bitová slova), kde první z nich nese synchronizační hlavičku a část dat (řídících nebo datových) a druhá obsahuje zbylou datovou část. Aby bylo možné zaznamenat, že byly zpracovány obě části, používá se proměnná `done`, která říká, že byla sekvence dokončena. V poslední řadě je v každé sekvenci řešeno generování `data_vld` signálu jednou za 32hodinových taktů a skramblování každého slova. Jak již bylo zmíněno, pro účely důkladnější verifikace RX PCS bloku bylo sestrojeno více typů těchto sekvencí. Jejich výčet a popis je pro jednoduchost zmíněn v tabulce 5.3. Všechny tyto typy jsou pak naskládány

Název	Popis
sequence_simple	Jednoduchá sekvence, jejímž úkolem je generování bezchybného provozu. Generuje dvojice 32bitových slov. První část obsahuje indikátor sekvence, který je definován tabulkou ve standardu IEEE 802.3. Druhá část pak obsahuje příslušná data, která mají následovat.
sequence_seq_err_inj	Oproti předchozí sekvenci, má tato za úkol vkládat do validních sekvencí chyby a to jmenovitě vadné sekvenční indikátory, které pak dekodér dekóduje jako sekvenční errory.
sequence_hdr_err_inj	Tato sekvence slouží ke generování chybných hlaviček, které mají za úkol na dekodéru vyvolat blokové errory. Pokud jich je patřičný počet, je vyvolán pád linky.

Tab. 5.3: Tabulka PMA sekvencí

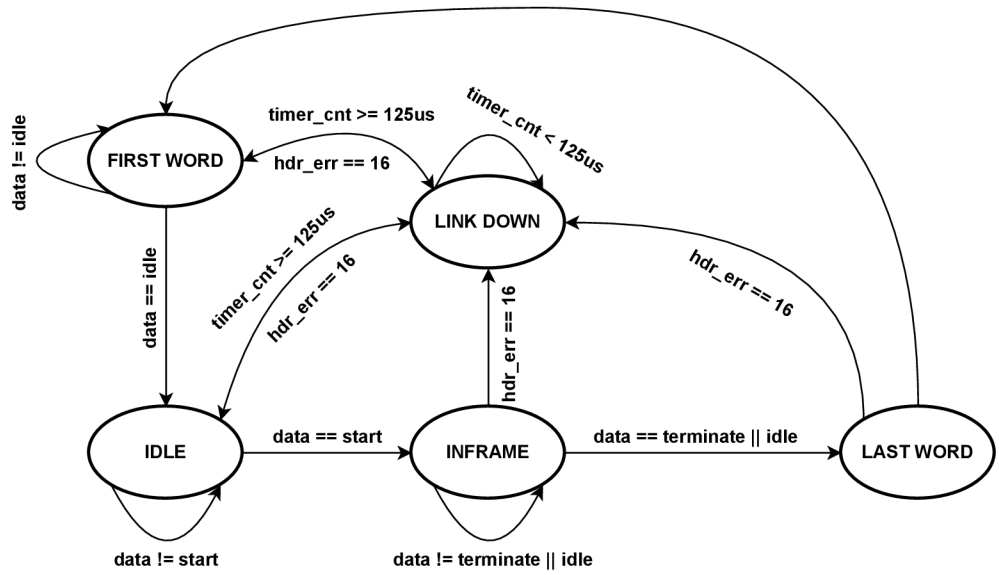
do PMA knihovny sekvencí. Aby byla možná synchronizace skramblovacího registru, `data_vld` signálu a počtu taktů mezi sekvencemi, byl vytvořen objekt `data_reg`, který příslušné informace obsahuje a sdílí mezi nimi.

5.8 Byte Array to PMA monitor

Aby bylo možné odesílat vysokoúrovňové rámce komponentě scoreboard, je nutné přijímat nízkoúrovňové PMA transakce na úrovni tohoto monitoru a na základě jednotlivých částí je skládat do Byte Array sekvence. K tomuto účelu implementuje metodu `write`, jejíž stavový diagram je zobrazen na obrázku 5.12 a v další části bude podrobněji popsán.

Pokud dojde k jejímu zavolání, spustí se sled funkcí, které se iterativně opakují. V prvé řadě se jedná o čítání času, které se využívá k zajištění minimální doby pádu linky. Jelikož mohou přicházet slova se špatnou hlavičkou, je další funkcionalitou následně s každým validním slovem kontrolováno, zda-li počítadlo chybných hlaviček nenabylo hodnoty 16, pokud tomu tak je, dojde k vyčištění datové paměti typu FIFO a dojde k pádu linky (nastavení stavu `LINK_DOWN`). Poslední takovou funkcí je deskramblování dat, které je potřeba provést, aby byla data čitelná.

V následujících částech již figuruje zmíněný stavový automat. Skládá se z 5 stavů, v každém z nich je prováděna kontrola validity hlaviček, pokud je nalezena nevalidní hlavička, dojde k inkrementaci čítače, v opačném případě je nastaven příznak



Obr. 5.12: Vývojový diagram funkce write Byte Array to PMA monitoru

na základě hlavičky buď na hodnotu 1 (datová), nebo 2 (kontrolní). Prvním stavem je `FIRST_WORD`, jehož účelem je naleznout první validní idle sekvenci, jež zapříčiní povolení čítače času, vložení první hlavičky a 4 datových bytů a přepnutí do stavu `IDLE`. Zde jsou vkládány přijaté hlavičky a data, zároveň probíhá kontrola sekvenčních indikátorů nacházejících se v prvním bytu transakce. V bezchybném případě se za nějaký čas na vstupu objeví transakce obsahující indikátor začátku rámce. V takovém případě dojde k přepnutí algoritmu do stavu `INFRAME`, kde jsou opět vkládány hlavičky, data příchozích transakcí a probíhá již zmíněná kontrola sekvencí, tentokrát je v příchozích transakcích hledán indikátor terminate označující konec rámce. Jeho nalezená zapříčiní přechod automatu do stavu `LAST_WORD`, ve kterém jsou vloženy poslední byty dat a ve formě Byte Array transakce jsou tato data zapsána na analysis port. Na závěr algoritmus přejde do počátečního stavu `FIRST_WORD` a iterace je opakována s příchodem další transakce. V jakémkoliv jiném případě přejde automat do stavu `LAST_WORD` a provede již zmíněné operace hned následující takt.

6 Testování

Tato kapitola pojednává o samotném testování hlavní komponenty a jejích částí. K verifikaci již zmíněného návrhu bude použit program ModelSim, jenž vznikl pod záštitou společnosti Mentor Graphics a hojně se využívá právě pro účely simulace a verifikace návrhů vytvořených v jazycích VHDL či Verilog. Dále zde bude uveden popis vytvořených komponent TestBech, nejvýše postaveného prostředí, patřičných modelů použitých na danou problematiku, scoreboardu, komponenty Test a jejich odlišností pro každý TestBench. Následně budou diskutovány simulační výsledky již zmíněné verifikace a naměřené hodnoty latence a přenosové rychlosti. V závěru pak budou shrnuty výsledky diplomové práce.

6.1 TestBench dílčích komponent

Tento modul slouží jako obálka propojující DUT s jednotlivými rozhraními, definuje jejich funkcionální podmínky a stará se o hodinovou doménu a resetování návrhu. Pro tuto verifikaci byly sestrojeny 3 typy, které jsou s menšími úpravami použity na testované komponenty. Každý TestBench je složen z jednotlivých sběrnic, kterými propojuje DUT s verifikačním prostředím a generátorů hodin a resetu. Jelikož ne všechny verifikované komponenty používají asynchronnní reset, je resetovací logika realizována dvojím způsobem. První možností je jednoduché generování resetu na začátku verifikace. Druhé řešení využívá znalosti zpoždění resetu a do návrhu tedy zavádí nezpožděný reset. Naopak verifikačnímu prostředí je přidělen zpožděný, aby byla možná synchronizace mezi těmito bloky. Druhá varianta je použita u top komponenty ETH PHY, kde je pro řešení asynchronního resetování vyveden z návrhu port `RESET_DONE`, jenž je použit jako indikátor spuštění zaznamenávání výstupních signálů z komponenty, tudíž není zapotřebí dodatečná resetovací logika. Další nezbytnou součástí tohoto modulu je komponenta Test, která v sobě ve většině případů obsahuje virtuální sekvenci, jež na základě sekvencí umístěných v sekvenční knihovně generuje data a metadata pro verifikovanou komponentu. Tento modul dále obsahuje top komponentu, jež propojuje nižší prostředí, které náleží daným verifikačním rozhraním. V poslední řadě se zde nachází scoreboard, který se liší pro každou komponentu a obsahuje porovnávací logiku, model, jenž je různý pro každou komponentu, a pro případ komponenty ETH PHY i jednotlivé měřící algoritmy. V dalších sekcích bude dále popsáno testování jednotlivých komponent a modely využité k jejich testování.

6.1.1 TX MAC

První verifikovanou komponentou, která tvoří první stupeň logiky ETH PHY, je TX MAC. Jejím úkolem je přebírat data generována uživatelem a doplňovat je o preambuli a SFD na začátku a hodnotou CRC na konci rámce. Její top level prostředí je složeno celkově ze tří dílčích prostředí. LII ENV na vstupu a výstupu, pro generování a kontrolu dat, a LII RDY ENV, k randomizaci RDY signálu. Nad těmito verifikačními komponentami pak sídlí TestBench, v němž jsou zapojeny vstupní a výstupní LII rozhraní. Jelikož komponenta nepoužívá asynchronní reset, bylo využito první varianty resetovací logiky. V této sekci bude dále popsán scoreboard a model, jenž definuje korektní chování komponenty. V poslední řadě budou vyhodnoceny získané výsledky.

Scoreboard a model

V této sekci bude uveden popis vyhodnocovací komponenty scoreboard a testovacího modelu. Scoreboard u této komponenty disponuje jedním výstupním datovým a dvěma vstupními porty, kde první z nich je datový a druhý obsahuje metadata ve formě indikátoru korupce CRC sekvence, který je následně využíván ve scoreboardu k porovnávání dat. Následné vyhodnocování se tedy provádí s ohledem na tento indikátor. Mohou nastat dvě situace. Buď má hodnotu logické 1, což způsobí ignorování příchozích dat a inkrementování čítače CRC chyb, nebo je nastaven do logické 0, čímž je indikováno, že má proběhnout porovnání vstupních dat s referenčními. Na základě předchozích kroků je pak verifikace vyhodnocena jako úspěšná nebo neúspěšná a její výsledek je oznámen na jejím konci.

Obr. 6.1: Vývojový diagram modelu TX MAC

Model slouží v tomto případě k simulování chování referenční komponenty TX MAC. Jeho funkcionalita lze popsat stavovým diagramem na obr. 6.1. Prvním krokem je vytažení dostupné transakce z portu TLM Analysis FIFO, z jehož dat je vypočítána hodnota referenčního CRC. Následně je vytvořena a inicializována referenční Byte Array transakce a na její začátek je vložen SFD indikátor a preambule,

za níž jsou umístěna data, která byla získána ze scoreboardu. Na konec této transakce je umístěna již zmíněná hodnota referenčního CRC. Takto vytvořená transakce je posléze zapsána na analysis port, kde si ho následně přebírá scoreboard.

Simulační výsledky

Po zkonstruování top level prostředí přichází na řadu testování, které bylo prováděno v již zmíněném nástroji ModelSim. Na obrázku 6.2 je možné vidět souhrn výsledků. Na první pohled je zjevné, že verifikace dopadla úspěšně, to znamená, že v průběhu nebyly nalezeny žádné chyby. Pro testování bylo vygenerováno 12139 různých typů vysokoúrovňových transakcí, díky nimž bylo možné vytvořit dostatečně velké množství různých typů nízkoúrovňových transakcí. Dále je patrné, že 2154 transakcí obsahovalo úmyslné vložení CRC chyby, které důkladně otestovalo tuto funkcionalitu.

```
# ------------------------------------------------
# ----  VERIFICATION SUCCESS      ----------------
# ------------------------------------------------
# ----  SCOREBOARD REPORT         ----------------
# ----  Count of added items:                12139
# ----  Count of removed items:               9985
# ----  Count of items DUT fifo:                 0
# ----  Count of items model fifo:              0
# ----  Count of CRC errors:                 2154
# ----  END REPORT                ----------------
```

Obr. 6.2: Souhrn výsledků verifikace komponenty TX MAC

Na konci verifikace bylo vygenerováno celkové pokrytí komponenty. Jak je zřejmé z obrázku 6.3, dosáhlo hodnoty 95.36 %, což je dostatečná hodnota na to, aby bylo možné prohlásit obvod za plně funkční. Zbylá procenta, která chybí do plného pokrytí, způsobily stavy komponenty, jež byly uvnitř návrhu definovány, ale nikdy však nesmí nastat. Výsledkem této verifikace, bylo tedy zjištění, že komponenta neobsahuje žádné chyby, které by musely být opraveny.

Coverage Summary By Instance (95.36%)

Instance	Assertions	Branches	Conditions	Expressions	Statements	Total
Total	100%	97%	90%	90.9%	98.9%	95.36%
DUT_U	100%	97%	90%	90.9%	98.9%	95.36%
lii_property	100%	-	-	-	-	100%

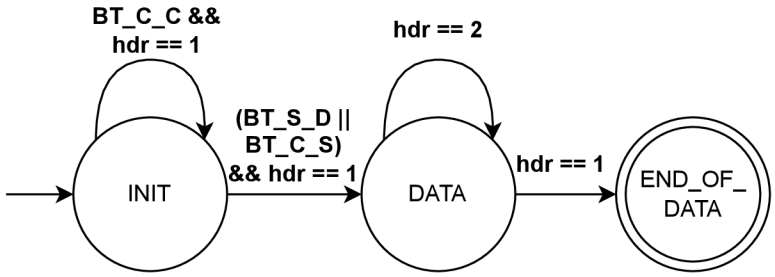
Obr. 6.3: Pokrytí komponenty TX MAC

6.1.2 TX PCS

Další testovanou komponentou, která představuje druhý stupeň logiky ETH PHY, je TX PCS, jejímž úkolem je provádět kódování a scramblování dat přijatých z TX MAC podvrstvy. K její verifikaci jsou použita virtuální rozhraní LII na vstupu a PMA na výstupu. Top level prostředí se tedy skládá z dílčích prostředí. LII ENV pro generování dat, LII RDY ENV, jež je využito pro signál RDY, a PMA ENV. Jelikož se v komponentě využívá asynchronní reset, je využita druhá varianta resetovací logiky, která byla popsána v sekci 6.1. Cílem této sekce je objasnit přístup k jejímu otestování. V následujících podkapitolách bude tedy popsán funkcionální model a scoreboard sestrojený pro tyto účely. Následně budou uvedeny výsledky verifikace a odhalené chyby.

Scoreboard

Pro účely porovnání přijatých dat z komponenty a referenčních dat z modelu byla vytvořena komponenta scoreboard. Pro tuto verifikaci byla pro ukázku použita druhá metoda pro ověření chování verifikovaného návrhu, kde se na testování pohlíží inverzním způsobem. Přesněji se jedná o to, že kodér, který se nachází na podvrstvě PCS, je verifikován připojením dekodéru, jehož logika je vytvořena právě ve scoreboardu. Tento dekodér má za úkol data z komponenty dekódovat podle stavového automatu na obrázku 6.4, a porovnat je s daty získanými z modelu.



Obr. 6.4: Stavový automat dekodéru

Procedura probíhající ve scoreboardu začíná načtením transakcí z DUT a modelu. Následně se v cyklu začnou procházet data transakcí z DUT a na základě jejich obsahu je vstoupeno do příslušného stavu stavového automatu. Před vstupem do jakéhokoliv stavu je z dat vždy načtena synchronizační hlavička a identifikátor dané sekvence, aby se neobjevovali v porovnávaných datech.

Počátečním stavem je `INIT`, v němž se očekává, že se na vstupu nacházejí mezipaketové mezery. Zde je kontrolována nejprve již zmíněná hlavička. Pokud se jedná o řídící hlavičku, kontroluje se co je obsaženo v identifikátoru. V opačném případě je

vyhlášena chybová hláška. Opětovný přechod do stavu `INIT` iniciuje značka `BT_C_C`. Z tohoto stádia přechází stavový automat do stavu `DATA` a to v případě, že se na vstupu objeví identifikátor začátku rámce. Mohou nastat dvě možnosti, buď transakce obsahuje sekvenci `BT_C_S`, v takovém případě dojde k useknutí preambule, idlu a k uložení prvních dat do paměti typu FIFO, nebo může transakce obsahovat identifikátor `BT_S_D`, čímž je řečeno, že součástí příští transakce už budou data. Pokud tedy dojde k přesunu automatu do stavu `DATA`, je započato ukládání dat do již zmíněné FIFO paměti a to v případě, že se na vstupu objevila správná hlavička. V okamžiku, kdy se na vstupu objeví jeden z identifikátorů označující konec rámce (jakýkoliv terminate), je na základě něj určen počet validních bytů a do paměti je uloženo poslední datové slovo. Na konci přejde automat do stavu `END_OF_DATA`. Ten signalizuje, že se nacházíme na konci cyklu a neměla by se objevit další data. Pokud se objeví, je vyvolána chybová hláška a verifikace se ukončí. V opačném případě je ověřeno, jestli tento stav nastal (to je z důvodu celistvosti rámce). Následně jsou data z FIFO paměti uložena do vysokoúrovňové transakce a porovnána. Pokud se verifikace dostala na konec svého běhu, je zavolána `report_phase`, ve které je vypsán výsledek.

Model

Pro účely úpravy vstupních dat, aby je bylo možné ověřovat ve scoreboardu, byl vytvořen model. Jeho funkcionalita je velice jednoduchá. Má za úkol pouze odebrat z přijatého rámce `SFD` sekvenci a preambuli. Takto upravená transakce je odeslána scoreboardu pomocí Analysis portu ke zpracování.

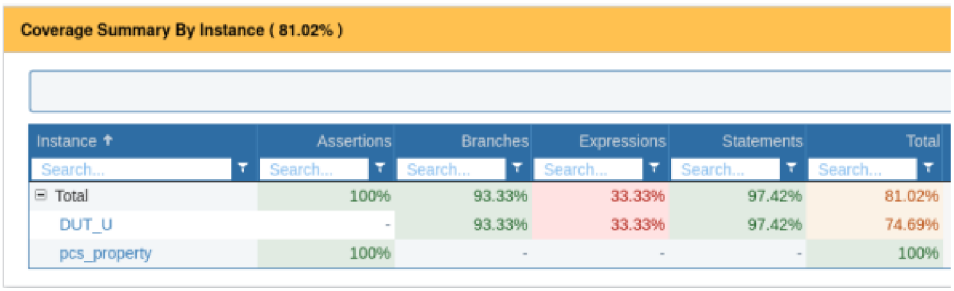
Simulační výsledky

Následujícím krokem je samotná verifikace. Obrázek 6.5 zobrazuje přehled výsledků. Na základě výpisu `VERIFICATION SUCCESS` je zřejmé, že verifikace proběhla úspěšně a během jejího běhu se nevyskytly žádné problémy. Bylo vygenerováno 15627 vysokoúrovňových transakcí, které umožnily vytvoření velkého množství LII transakcí, díky nimž bylo provedeno důkladné testování verifikovaného návrhu.

```
----------------------------------------------------
----  VERIFICATION SUCCESS      -------------------
----------------------------------------------------
----  SCOREBOARD REPORT         -------------------
----  Count of added items:           15627   ----
----  Count of removed items:         15627   ----
----  Count of items DUT fifo:            0   ----
----  Count of items model fifo:          0   ----
----  END REPORT                -------------------
```

Obr. 6.5: Souhrn výsledků verifikace komponenty TX PCS

Jak je zřejmé z obrázku 6.6, verifikace pokryla svým testováním 81 % verifikovaného návrhu. Nižší pokrytí bylo způsobeno položkou Expression, která dosahuje pouze 33 %, tato hodnota se pojí s komponentou generující asynchronní reset, která obsahuje generické parametry, jež nejsou v tomto návrhu používány, tudíž nedojde k aktivaci některých částí kódu. Další problematickou komponentou je scrambler, který je vytvořen pro délky až 64 bitů. Jelikož se v tomto případě využívá pouze 32bitová sběrnice, nedošlo ke kompletnímu pokrytí této komponenty. Z předchozích poznatků bylo tedy zjištěno, že pokrytí komponenty by bez těchto negativních vlivů bylo daleko větší. Komponenta tedy byla důkladně otestována a během této procedury nebyly objeveny žádné chyby, které by bylo nutné opravit.



Obr. 6.6: Pokrytí komponenty TX PCS

6.1.3 RX PCS

Další testovanou komponentou je RX PCS. Jak již bylo řečeno v teoretické části, má za úkol provádět descramblování a dekódování dat přijatých z vrstvy PMA. Její TestBench se skládá z rozhraní PMA na vstupu a LII na výstupu, skládá se tudíž z dílčích prostředí PMA a LII ETH PHY. Jelikož je zde využíván asynchronní reset, byla opět zvolena druhá varianta resetovací logiky. V této kapitole bude dále popsáno, jakým způsobem bylo přistupováno k jejímu testování, zejména funkcionalita použitého modelu a scoreboardu. Na závěr sekce budou diskutovány získané výsledky a odhalené chyby.

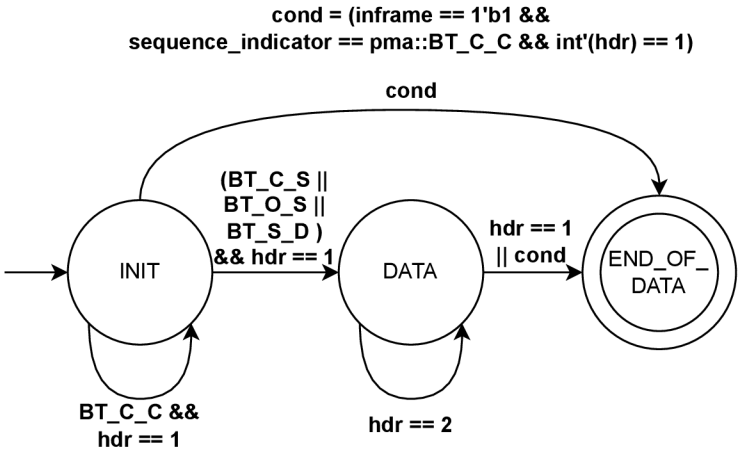
Scoreboard

Jak již bylo zmíněno, tato komponenta slouží k shrnutí a porovnání získaných výsledků. Stejně tak, jak tomu bylo u první komponenty, i zde bude použit klasický přístup testování. Scoreboard v tomto případě postupně odebírá vysokoúrovňové a meta transakce z portu typu TLM FIFO. Ty jsou následně porovnány v `run_phase` s referenčními transakcemi pocházejícími z modelu. Na základě této funkcionality je

posléze v případě neshody vyhlášena patřičná chybová hláška. Na závěr celé verifikace je, jak tomu bylo již u předchozích případech v `report_phase`, proveden výpis výsledků verifikace.

Model

Pro účely vytvoření referenčních transakcí byl sestrojen model vrstvy RX PCS, který vychází ze standardu. Z hlediska funkcionality jej lze opět popsat stavovým automatem (obr. 6.7). Na první pohled se zdá být podobný předchozímu, ale liší se logikou nacházející se uvnitř jednotlivých stavů a přechody mezi nimi. Logika prochází příchozí Byte Array sekvenci po devíticích bytů (dále již transakce), kde první je vždy hlavička a dalších 8 jsou užitečná data, ve kterých vyhledává specifické identifikátory, na jejichž základě prochází jednotlivými stavy.



Obr. 6.7: Stavový automat modelu RX PCS

Počátečním stavem je `INIT`, v němž je nejprve zkontrolováno, zda-li transakce obsahuje kontrolní hlavičku. Následně probíhá kontrola indikátorů, kde mohou nastat 2 možnosti. Buď obsah transakce začíná kontrolním bytem s označením `BT_C_C`, v takovém případě algoritmus ignoruje datovou část a zůstává ve stejném stavu, nebo nastává situace, kdy se v některé z příchozích transakcí objeví jeden ze tří typů sekvencí indikující začátek rámce. Prvními z nich jsou `BT_C_S` a `BT_O_S`, které způsobí odseknutí preambule a zapsání příslušných bytů do paměti. Druhou možností je typ `BT_S_D` označující transakci, která mimo indikátor obsahuje i 7 validních bytů dat, jenž jsou zapsány do paměti, aby mohly být následně připojeny k dalším datům. Pokud nenastane v tomto ani jedna z dosud zmíněných situací, dojde k vyhlášení erroru. Jelikož se algoritmus nachází v `INIT` stavu, není jej zapotřebí ukončit a kontrola může pokračovat.

Po nalezení začátku rámce algoritmus přechází do stavu `DATA`, kde jsou datové transakce vkládány do paměti typu FIFO. Tato operace je opakována, dokud jsou v transakcích obsaženy datové hlavičky. Posledním krokem algoritmu je nalezení konce rámce. Tato operace je iniciována kontrolní hlavičkou na pozici prvního bytu transakce. Jak je patrné z obrázku 3.5, existuje 8 možností zakončení rámce. Každá z nich má specifický identifikátor a udává počet validních datových bytů, proto i model zohledňuje tuto skutečnost a na jejich základě ukládá pouze patřičný počet bytů do paměti. Pokud v tomto stavu nastane výše nespecifikovaná situace, dojde k nastavení bitu iniciujícího chybný stav, inkrementování čítače chybných rámců a ukončení algoritmu. V každém případě, který v tuto chvíli může nastat, přejde stavový automat do stavu `END_OF_DATA`. V tuto chvíli by již měla být Byte Array transakce prázdná, a proto by se v tomto stavu neměla vykonat žádná logika. V opačném případě skončí verifikace fatal errorem. Na závěr celého algoritmu jsou zapsána data z paměti do výstupní Byte Array transakce, která je posléze odeslána po analysis portu do scoreboardu. Tento algoritmus je opakován pro každý další příchozí rámec.

Simulační výsledky

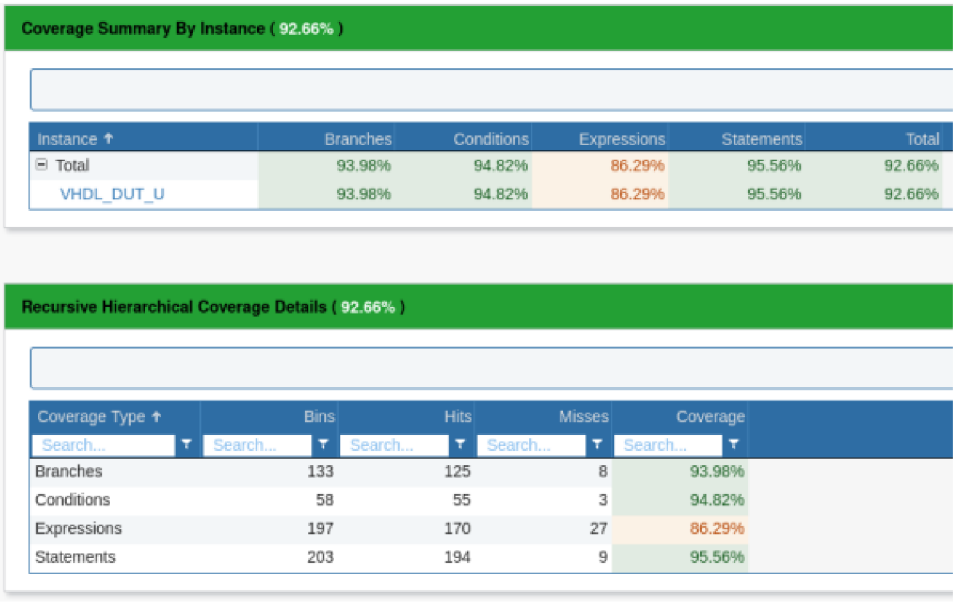
Nyní bude pozornost zaměřena na výsledky získané verifikací zmíněné komponenty. Jak je patrné z obrázku 6.8, testování proběhlo úspěšně. Na vstup DUT bylo vygenerováno 13599 Byte Array transakcí, z čehož 3442 bylo vygenerovaných s úmyslnou chybou, v tomto případě se jednalo buď o vygenerování sekvenčních errorů, či úmyslného shození linky. Verifikací byly ověřeny i okrajové případy, díky jimž byly objeveny chyby v návrhu, na které se zaměříme v sekci 6.1.3.

```
| ------------------------------------------------
| ----  VERIFICATION SUCCESS      ----------------
| ------------------------------------------------
| ----  SCOREBOARD REPORT         ----------------
| ----  Count of added items:           13599  ----
| ----  Count of removed items:         13599  ----
| ----  Count of dropped items:          3442  ----
| ----  Count of items DUT fifo:            0  ----
| ----  Count of items model fifo:          0  ----
| ----  DUT used:                           0  ----
| ----  Model used:                         0  ----
| ----  END REPORT                ----------------
```

Obr. 6.8: Souhrn výsledků verifikace komponenty RX PCS

Dalším výsledkem verifikace bylo pokrytí verifikované komponenty. Jak je patrné z obrázku 6.9, testování pokrylo 92.66 % stavů, které v obvodu mohou nastat. Chybějící procenta pokrytí souvisí v prvé řadě s komponentou descrambler, jež nemohla být důkladně otestována, protože je vytvořena pro rozsah délek 32 až 64 bitů. Jelikož v návrhu je používaná kratší varianta, druhý stav nemohl být ověřen. Druhou problematickou komponentou je asynchronní reset, jenž obsahuje množství parametrů,

které nejsou v návrhu používány, a tak příslušné části kódu nebyly ověřeny. Závěrem bylo tedy zjištěno, že komponenta byla důkladně otestována a může být zařazena do provozu.



Coverage Summary By Instance (92.66%)

Instance	Branches	Conditions	Expressions	Statements	Total
Total	93.98%	94.82%	86.29%	95.56%	92.66%
VHDL_DUT_U	93.98%	94.82%	86.29%	95.56%	92.66%

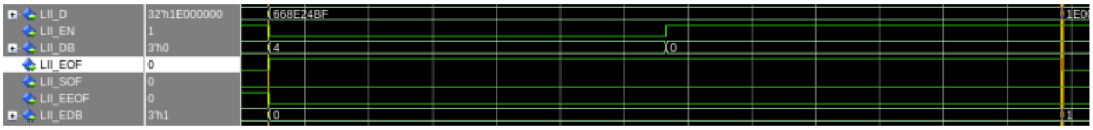
Recursive Hierarchical Coverage Details (92.66%)

Coverage Type	Bins	Hits	Misses	Coverage
Branches	133	125	8	93.98%
Conditions	58	55	3	94.82%
Expressions	197	170	27	86.29%
Statements	203	194	9	95.56%

Obr. 6.9: Pokrytí komponenty RX PCS

Odhalené chyby

Jak již bylo zmíněno v předchozí kapitole, nyní budou diskutovány zjištěné chyby. První z nich bylo špatné chování signálu `bytes_vld` při pádu `RDY`. Verifikace tuto chybu objevila porovnáním referenční transakce s výstupním rámcem získaným z DUT. Tuto skutečnost znázorňuje výpis z transcriptu simulačního nástroje ModelSim na obrázku 6.18.



Obr. 6.10: Chyba při nastavení validních bytů wave

Chyba byla následně zachycena i v waveformu, jenž je zobrazen na obrázku 6.19. Zde je již vidět problém, který odporuje standardu. Nejdříve se na sběrnici objeví signál `EOF` spolu s počtem validních bytů `LII_DB` a signálem `RDY` v logické 0. Dále se však již hodnota `LII_DB` nepropaguje do dalších taktů, čímž dojde v komponentě ke špatnému přiřazení dat na výstup a verifikace tím pádem selže.

Obr. 6.11: Chyba při nastavení validních bytů

Další zjištěnou chybou byla špatná funkčnost komponenty `BER_MONITOR`, která se stará o počítání chybných hlaviček. Na základě jejich počtu je nastavován signál `HI_BER`, jenž je jedním z parametrů určující stav linky. K vykonávání této funkce monitor obsahuje čítač, který je inkrementován s každou validní PMA hlavičkou. Jeho hodnota posléze definuje minimální časový interval, po který má být linka shozena. Jak hovoří teorie, mělo by se jedna o interval minimálně 125 μs. Pokud však chybovost přetrvává, je linka opakovaně shazována. Chyba, jež byla v komponentě nalezena, souvisí právě s tímto čítačem.

Obr. 6.12: Stav čítače a linky na začátku chyby

Na obrázku 6.12 je zobrazena situace, kdy se na Ethernetové vrstvě projevila taková chybovost, že došlo k nastavení signálu `HI_BER` do logické 1 a pádu linky. Dále je zde také patrná aktuální hodnota čítače. Následující obrázek 6.13 zobrazuje chybu zachycenou v transcriptu, z něhož je patrné, že na výstupu DUT se objevila odlišná transakce od referenční.

Důvod tohoto konfliktu zobrazuje obrázek 6.14, na kterém je vidět předčasné přetečení čítače a následné nahození linky. Korektní chování se váže k jeho počáteční hodnotě. Pokud je nulová, čítač dopočítá do maximální hodnoty a dojde k nahození linky. V opačném případě, poté co přeteče, napočítá opět do jeho maxima a následně je na základě signálu `HI_BER` zjištěno, zda-li se na lince vyskytují chyby. Pokud je

```
UVM_FATAL /home/local/xkrizd01/rx_pcs_fix/fwbase/projects/hft/comp/eth_phy/10ge/pcs/uvm/rx/tbench/env//scoreboard.sv(95) @ 1227369925
 The items does not match.
 Model data:
      Byte_array::sequence_item size 3537
              78 fa c1 59 29 ce 4c 64    4f 97 8a 92 f5 8c df 49    80 a1 e8 4a 6c 88 9a 13    6b b3 89 71 05 b3 d3 e1
              09 81 9e d9 c4 5d f3 df    51 12 48 77 28 e5 8e 34    ec d3 c2 8c 0e 55 13 8f    c8 14 e6 d8 ec e4 66 ed
              2b 13 b9 3a 79 02 66 22    01 87 ff 13 cb 5a 86 82    e1 d4 65 a4 2e dd bd 0d    02 08 91 77 d0 cf 3f f1
              7a 13 a4 38 ff e1 58 e3    21 f9 f8 bf a5 30 38 9c    d7 a6 03 e4 13 26 a3 fb    dc 0a 0d 1f 27 7a 48 57
 DUT data:
      Byte_array::sequence_item size 3533
              78 d1 0d 0f e7 71 05 27    ad d9 bc 5a f3 66 0a 0c    f6 8d db 9b 60 72 3d db    e3 48 ed 1f 37 77 2b af
              88 1f 57 60 e2 01 5b 23    68 a6 22 e2 db 2c b7 1a    6b 8d 85 db e1 07 bb 5b    77 ca b0 5d 6a 87 e4 9c
              63 6e 1c bd 85 dd 23 57    92 50 3a 6a 02 bb 1b 7f    80 9c ec 1a d6 93 e3 24    a6 a6 a6 d0 17 12 70 2d
              0a 99 83 33 35 11 b4 89    bc a4 ca ac 87 65 be a7    22 c0 93 0a 82 82 da d4    bd 97 9d 68 29 4c c7 3e
```

Obr. 6.13: Chyba při shazování linky

jeho hodnota v logické 1, situace se opakuje. V opačném případě dojde k nahození linky a vynulování čítače. Zmíněné chyby byly nahlášeny návrháři tohoto obvodu, opraveny a posléze opět otestovány.

Obr. 6.14: Stav čítače a linky na konci chyby

6.1.4 RX MAC

Poslední dílčí komponentou je RX MAC. Jak již bylo popsáno dříve, tato komponenta má za úkol koncové zpracování LII transakcí z dekodéru. Jejím úkolem je odsekávat a kontrolovat nadbytečná data, kterými je preambule se SFD indikátorem a CRC sekvence. V poslední řadě se stará o oznamování vzniklých chyb uživateli. TestBench je zde obdobný jako pro TX MAC v sekci 6.1.1, proto nebude znovu popisován. Dále v této kapitole budou opět popsány verifikační komponenty scoreboard a model. Na závěr budou debatovány získané výsledky a odhalené chyby.

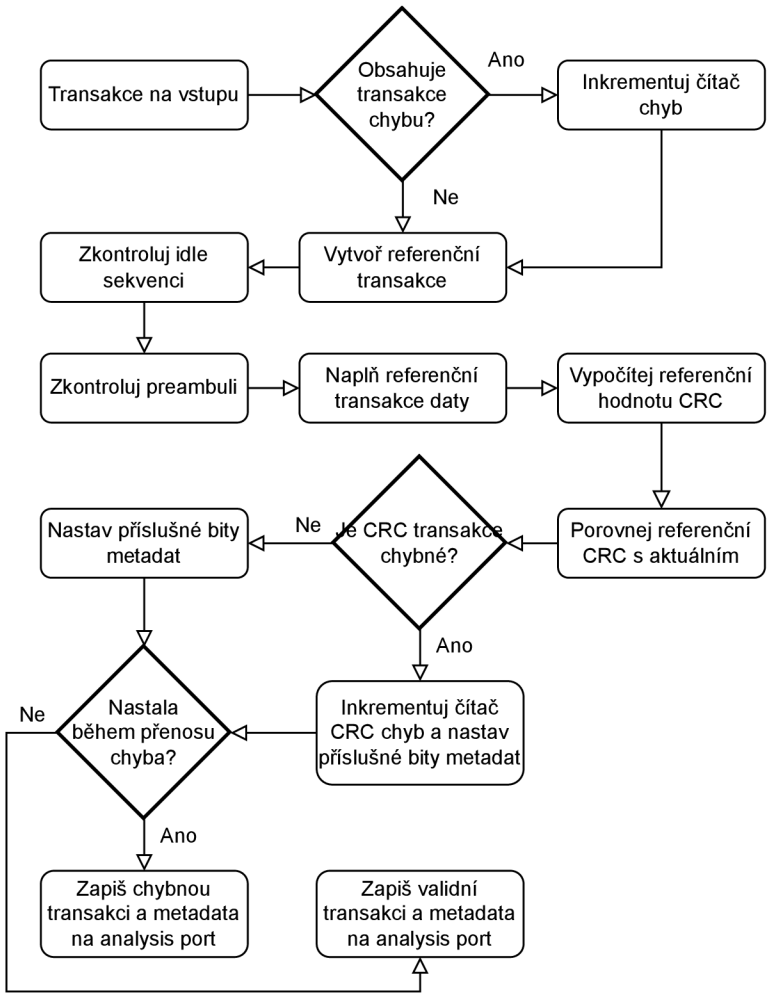
Scoreboard

První zmíněnou komponentou je scoreboard, jež v tomto případě sestává ze 2 vstupních a 2 výstupních portů, které jsou využívány k získání vstupních a výstupních dat a metadat, na jejichž základě je posléze prováděno porovnávání. Prvním krokem tohoto procesu je opět získání transakcí z modelu a z DUT. Následně je provedena kontrola metadat, kdy je na základě pozice bitu inkrementován patřičný chybový čítač. Na prvním bitu je umístěn stav linky, na druhém pak údaj o chybě v sekvenci a na třetím a čtvrtém indikátory validity CRC sekvence. Tyto hodnoty jsou posléze

porovnávány s metadaty, které vyprodukoval model. Další částí je samotná kontrola datových transakcí, jež vyvolá chybnou hlášku, pokud se transakce neshodují a inkrementuje čítač chyb. Na závěr je pak v report fázi verifikačnímu technikovi oznámeno, zda-li verifikace proběhla úspěšně.

Model

Druhou důležitou komponentou je model. V tomto případě řeší generování referenčních transakcí, které by měl správně vyprodukovat RX MAC. Jeho funkcionalitu lze popsat vývojovým diagramem, jenž je zobrazený na obr. 6.15.



Obr. 6.15: Vývojový diagram modelu RX MAC

Vykonávání algoritmu započne příchodem transakcí na vstupy modelu, následně je na základě metadat zkontrolováno, zda-li neobsahuje error. Pokud ano, dojde k inkrementaci patřičných čítačů chyb. Ať chybu obsahuje nebo neobsahuje, algoritmus pokračuje dále a to vytvořením referenčních transakcí o patřičné délce. Jsou

zde definované dva typy. První z nich je validní, která obsahuje korektní data, jež jsou zakončená správným počtem bytů. Druhým typem je tzv. error transakce, kde jsou validní všechny poslední byty. Toto rozdělení je zde, aby bylo možné porovnat i chybné transakce, které jsou komponentou taktéž odesílány uživateli ke zpracování.

Dalším krokem algoritmu je odebrání nadbytečných dat a jejich porovnání s referenční hodnotou. Jedná se o preambuli, `SFD` indikátor a hodnotu CRC. Na závěr jsou na základě předchozích výsledků vyplněny bity metadat a příslušné transakce jsou zapsány na analysis port, kde jsou přebrány scoreboardem ke zpracování.

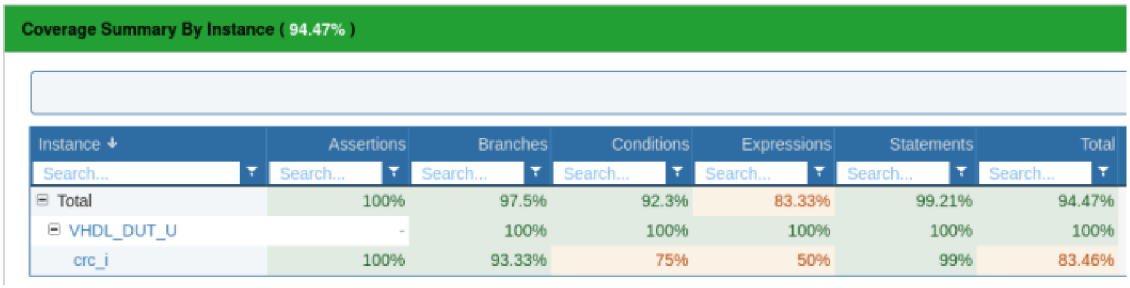
Simulační výsledky

S pomocí výše zmíněných funkcionalit byla provedena zevrubná verifikace komponenty RX MAC. Jak je patrné z výpisu VERIFICATION SUCCESS na obrázku 6.16, testování proběhlo úspěšně. Na vstup komponenty bylo vygenerováno 11919 Byte Array transakcí, které důkladně prověřily velké množství stavů, do kterých se komponenta může dostat. Na obrázku je dále možné vidět i výpis čítačů chyb. Nejsou to chyby, které by značily nefunkčnost komponenty, ale jedná se o errory, které se mohou objevit na Ethernetové lince. Jejich vkládáním do transakcí bylo ověřeno, jak se bude RX MAC chovat v případě, že na něm bude vysoká chybovost. Dále byl navozen stav shození linky a úmyslné vkládání CRC chyb. Díky tomuto testování bylo možné odhalit okrajové případy, ve kterých se komponenta nechovala správně, tyto chyby budou debatovány v následující sekci.

```
----------------------------------------------------
----  VERIFICATION SUCCESS      -------------------
----------------------------------------------------
----  SCOREBOARD REPORT         -------------------
----  Count of added items:                  11919
----  Count of removed items:                11919
----  Count of items DUT fifo:                   0
----  Count of items model fifo:                 0
----  Count of error items DUT fifo:          2264
----  Count of error items model fifo:        2264
----  Count of link down model:                710
----  Count of link down DUT:                  710
----  Count of model CRC error:               1466
----  Count of DUT CRC error:                 1466
----  END REPORT                -------------------
```

Obr. 6.16: Souhrn výsledků verifikace komponenty RX MAC

Po úspěšně proběhlé verifikaci bylo vygenerováno pokrytí, aby bylo zjištěno, jak moc kvalitně byl obvod otestován. Jak je patrné z obrázku 6.17, testováním bylo pokryto 94.47 % verifikovaného návrhu. Zbylá procenta, která nebyla pokryta souvisí s komponentou pro generování CRC. Ta obsahuje části kódu, do kterých se v tomto případě nikdy nevstupuje, protože to komponenta neumožňuje.



Obr. 6.17: Pokrytí komponenty RX MAC

Odhalené chyby

První nalezenou chybou bylo špatné vyhodnocování hodnoty CRC, jejíž odhalení je zobrazeno na obr. 6.18. Jak je patrné, neshodují se bity metadat, které obsahují hodnoty signálů `CRC_VLD` a `CRC_OK`, proto verifikace skončí fatal errorem.

```
CRC in meta is not same
DUT CRC VLD 1 and CRC OK 0
Model CRC VLD 1 and CRC OK 1
UVM_FATAL /home/local/xkrizd01
```

Obr. 6.18: Chyba při kontrole CRC transcript

Prvopočátek této chyby se nachází v okamžiku, kdy se v návrhu začnou objevovat sekvenční errory. V tento moment by mělo dojít k resetování hodnoty CRC, bohužel se tak nestane, protože hodnota `crc_rst`, která tento reset iniciuje, je v nedefinované hodnotě. Chybu je možné zpozorovat na obrázku 6.19.

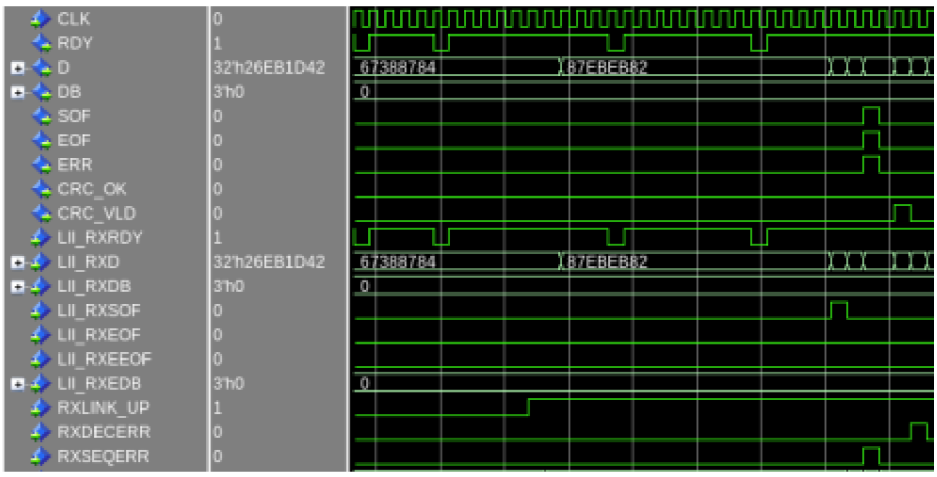
Obr. 6.19: Špatné přiřazení do signálu crc rst

Druhou nalezenou chybou bylo špatné reagování signálu `SEQERR` při generování preambule a `SFD` indikátoru. Obrázek 6.20 zobrazuje, jakým způsobem byla chyba odhalena verifikací. Je z něj patrné, že se neshodují metadata, zejména bity, které se týkají errorů.

Obrázek 6.21 zobrazuje chybu zobrazenou ve wavu. Lze z něj vypozorovat nekorektní chování sběrnice, specificky se jedná o nastavení signálů `SOF` a `EOF` zároveň. Správně by měl být sekvenční error ignorován, jelikož se chyba nevyskytla uvnitř datové části příslušného rámce.

```
# CRC in meta is not same
# DUT CRC VLD 1 and CRC OK 0
# Model CRC VLD 1 and CRC OK 1
# Wrong metadata          29377236698
# ERR DUT: 1, LINK STATUS model: 0, RXDECERR model: 0, RXSEQERR model: 0
```

Obr. 6.20: Chyba při nastavení SEQERR



Obr. 6.21: Chyba při nastavení SEQERR wave

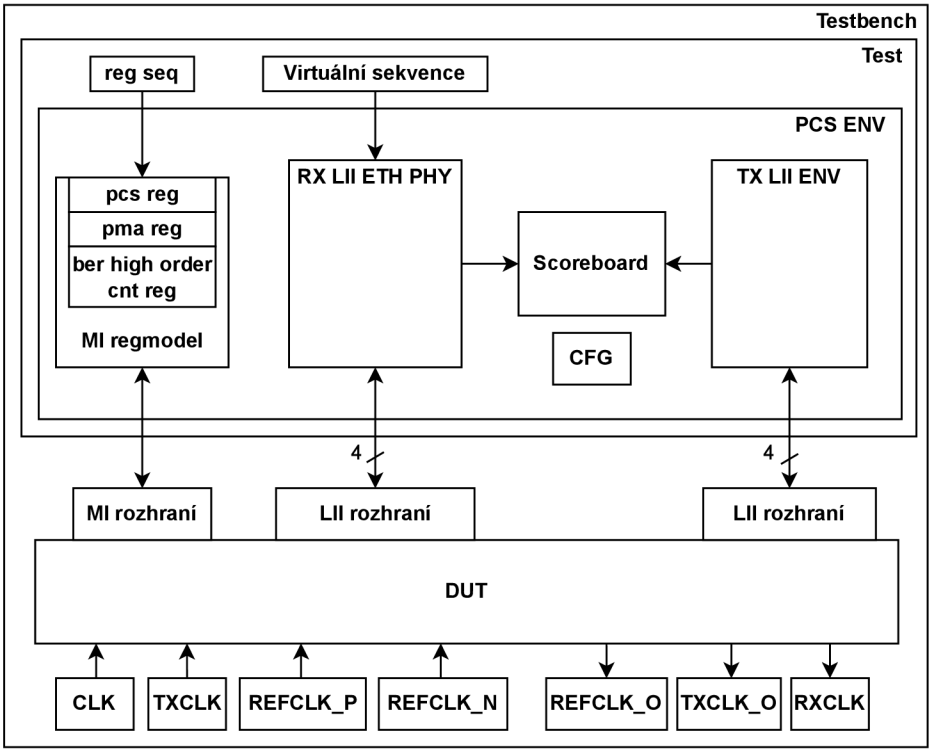
6.2 Verifikace fyzické vrstvy Ethernetu

Tato kapitola se zaobírá problematikou verifikace nízkolatenční fyzické vrstvy Ethernetu, zejména zde budou diskutovány jednotlivé verifikační komponenty, kterými je TestBench, scoreboard a model. Na závěr budou uvedeny výsledky verifikace a měření latence a propustnosti. Fyzická vrstva Ethernetu se skládá dílčích komponent TX MAC, TX PCS, RX PCS a RX MAC, jejichž zevrubná verifikace byla prováděna v předchozích krocích. Nyní bude důraz zaměřen na celkové chování této vrstvy.

6.2.1 TestBench

Základ verifikace fyzické vrstvy Ethernetu je tvořen TestBenchem, jenž znázorňuje obr. 6.22, který se skládá ze tří typů verifikačních rozhraní. Na levé části schématu se nachází MI rozhraní, k jemuž je připojen příslušný registrový model sloužící k ovládání zápisů a čtení na této sběrnici, bližší informace budou sděleny v jedné z dalších sekcí. Za účelem generování příslušných dat je na tento model přivedena registrová sekvence. Dále se skládá ze vstupního a výstupního LII rozhraní, která jsou zapojena do prostředí RX LII ETH PHY a TX LII ENV. O generování dat a metadat

prvnímu z těchto rozhraní se stará virtuální sekvence. V poslední řadě se zde nachází opět scoreboard pro závěrečné vyhodnocení verifikace.



Obr. 6.22: Testbench ETH PHY

Další důležitou částí tohoto testbenche je generátor hodin a resetu. Existují zde 4 vstupní a 3 výstupní hodinové domény. `REFCLK_N` a `REFCLK_P` jsou využity jako hodiny na základě, kterých GTY generují veškeré výstupní hodiny, jejich frekvence je 322.265625 MHz. Dále jsou zde `CLK`, jenž běží na frekvenci 125MHz a slouží jako hodiny, na kterých běží GTY. Posledním hodinovým vstupem jsou `TXCLK`, tento vstup je připojen přímo na výstupní hodiny `TXCLK_O`, a využívá je TX MAC ke generování provozu. Dále je zde výstupní hodinová doména `REFCLK_O`, která je pouze pro kontrolu správnosti. Posledním výstupem jsou `RXCLK`, které byli zjištěny pomocí PLL bloků, které slouží právě k rekonstrukci vstupních hodin, aby byla umožněna synchronizace mezi TX a RX stranou. Dále je zde realizována resetovací logika a to podle třetího typu resetu, který byl popisován v sekci 6.1 a to za pomoci výstupů `RXRST_DONE` `TXRST_DONE`.

Registrový model

Jak již bylo zmíněno verifikace disponuje tzv. registrovým modelem. Jedná se možnost pomocí níž, lze v metodologii UVM zapisovat do registrů, které se nacházejí

uvnitř verifikované komponenty. V této verifikaci je definován třemi komponentami. První z nich se nachází v souboru `registers.sv`, ten obsahuje seznam registrů `UVM_REG`, do kterých bude prováděn zápis, či se z nich bude číst. V tomto případě se jedná o registry `pma_control`, kterým lze ovládat blok PMA, lze jím například vyvolat reset daného bloku, zvolit přenosovou rychlost, či nastavit lokální smyčku na této vrstvě. Dalším je `pcs_control`, jenž se využívá k ovládání bloku PCS, umožňuje též vyvolat reset, či nastavit lokální smyčku, ale navíc je zde například možnost zastavení hodin pomocí položky `clk_stop_en`. Posledním registrem je `ber_high_order_cnt`, který disponuje pouze jednou položkou a to je BER čítač. Umožňuje pouze vyčtení jeho aktuální hodnoty, čímž by mělo být provedeno i jeho vynulování.

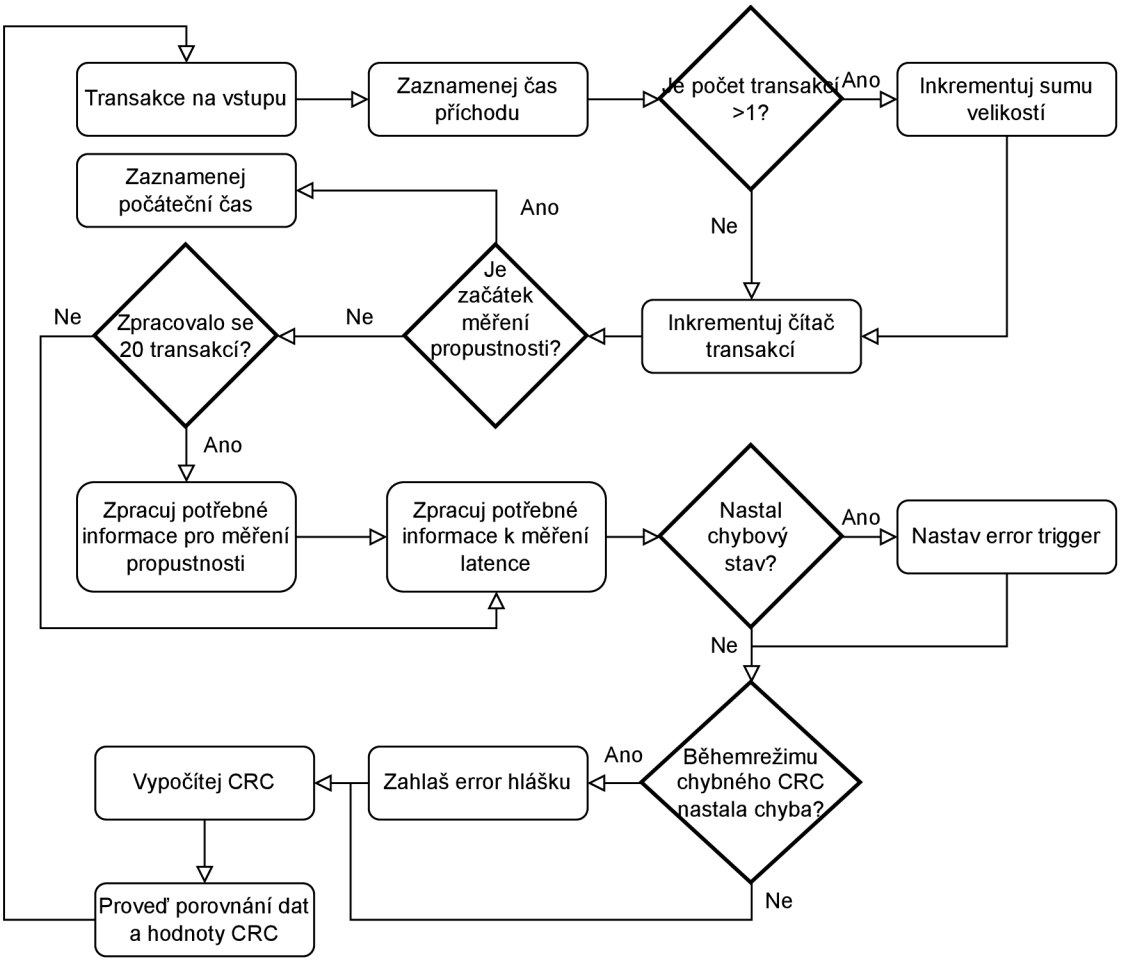
Druhou komponentou, která se k tomuto model váže je `UVM_REG_BLOCK`, která se nachází v souboru `regmodel.sv`. Jejím účelem je sjednotit registry do jednoho bloku a provést počáteční konfiguraci, tím je myšleno nastavení počáteční adresy, případně jejího offsetu a nastavení správného příznaku, který říká, jestli lze z registru jen číst, nebo jej lze použít i pro zápis. Poslední krokem je přidělení způsobu přístupu. V tomto případě je zvolen frontdoor, kdy je používáno specifické rozhraní, na které se váže časování. Mimo zmíněný přístup existuje i backdoor, jenž využívá databázi simulátoru a všechny operace s registry jsou vykonávány s nulovým simulačním zpožděním.

Poslední komponentou tohoto modelu je samotná sekvence, která pouze říká jakým způsobem se mají nastavovat bity samotných registrů. V tomto případě je zde definováno čtení i zápis některých bitů a jednotlivé operace jsou odděleny časovými okny.

Scoreboard

Scoreboard ETH PHY lze rozdělit do dvou částí. První z nich je `run_phase`, která je popsána vývojovým diagramem na obrázku 6.23, definuje kontrolu referenčních transakcí s výstupními a realizaci měření propustnosti a latence. Celý algoritmus začíná příchodem transakcí na jednotlivé porty typu TLM FIFO. Po odebrání transakcí z výstupu komponenty je zaznamenán čas, který je posléze využíván k výpočtu latence.

Dalším krokem je výpočet propustnosti. V této fázi je po příchodu první transakce aktivován příznak říkající, že měření začalo a je uložen čas začátku měření. Ve stejný okamžik je započato průběžné inkrementování velikosti přenesených dat a počtu transakcí. V okamžiku, kdy je počet přijatých transakcí dostatečný na to, aby bylo možné změřit propustnost, přejde algoritmus k jejímu výpočtu. Zde je nejprve zaznamenán čas konce měření a následně je ze získaných dat vypočítána propustnost podle vztahu: $throughput = \frac{velikost\ dat\ v\ bitech}{čas\ konce\ měření\ -\ čas\ začátku\ měření}$. Dále



Obr. 6.23: Scoreboard ETH PHY

je určena maximální a minimální propustnost a to tak, že se vždy porovná současná hodnota s předchozí. V dalším kroku se v případě výskytu transakce s jinou délkou než byli předchozí vypočítá průměrná hodnota propustnosti podle vztahu: $average\ throughput = \frac{suma\ propustností}{počet\ transakcí\ stejné\ délky}$. Na závěr jsou získaná data o propustnosti uložena do příslušných polí, aby je bylo posléze možné použít pro vygenerování souborů.

Následující rutinou je výpočet latence. Ten se provádí průběžně pro každou transakci podle vztahu: $delay = čas\ přijetí\ transakce\ -\ čas\ odeslání\ transakce$. Následně je opět obdobným způsobem jako u propustnosti zjišťována maximální, minimální a průměrná latence. Na závěr jsou opět uloženy hodnoty pro zápis do souborů. Poslední částí je pak kontrola dat a chybových stavů. Zde je v případě, že nastal některý z chybových stavů inkrementován patřičný čítač. Dále následuje kontrola metadat, správnosti vkládání CRC chyb prostým porovnáním metadat z modelu a příznaku CRCERR ze vstupu. Na závěr je zde kontrola samotného CRC a dat,

kde je v případě neshody ukončena verifikace vyhlášením FATAL erroru.

Poslední částí celého scoreboardu je report fáze. Oproti předchozím komponentám je zde volitelně na konci celé verifikace prováděn zápis jednotlivých hodnot do soubor, které jsou pak dostupné uživateli k nahlédnutí. Na závěr jsou pak vypsány verifikační výsledky, které budou blíže specifikovány v sekci 6.2.2.

Model

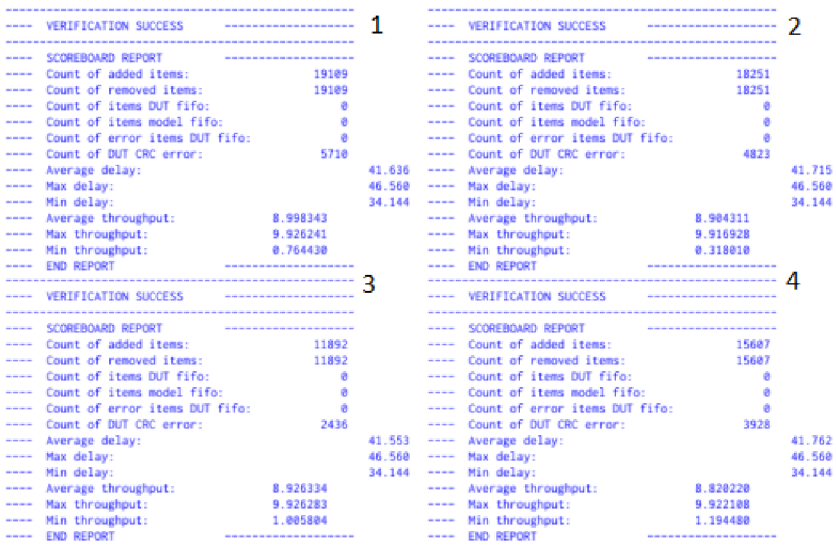
Model této komponenty je, jak z vývojového diagramu na obrázku 6.24 vyplývá, velice jednoduchý. Nejprve je v momentě, kdy se transakce objeví na vstupu portu TLM FIFO, zaznamenán čas, který je posléze využíván k výpočtu latence celého obvodu. Následně se z přijatého rámce vypočítá hodnota CRC, která je společně s daty a metadaty zapsána na výstupní analysis porty.

Obr. 6.24: Model ETH PHY

6.2.2 Simulační výsledky

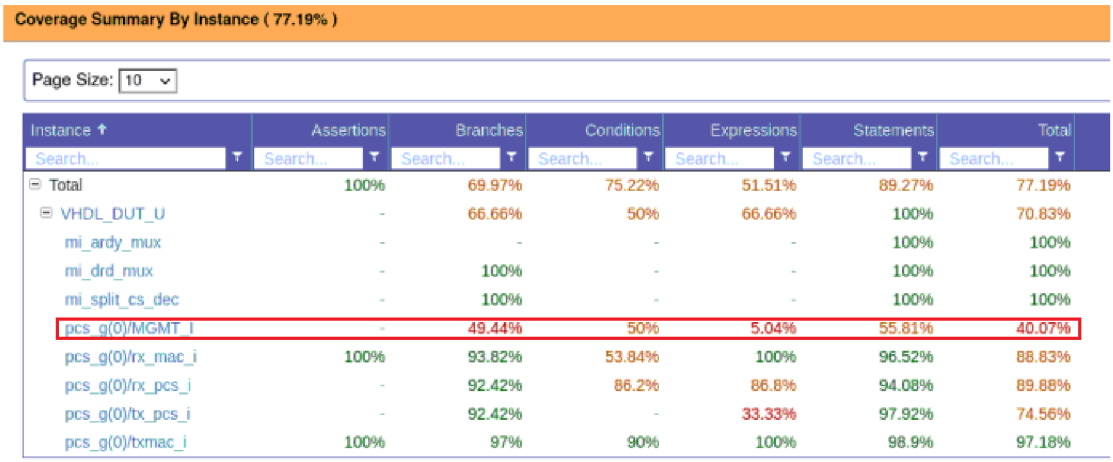
Tato kapitola bude věnována výsledkům, které byly zjištěny během verifikace nízkolatenční fyzické vrstvy Ethernetu. Na obr. 6.25 je zobrazeno shrnutí výsledků verifikace všech 4 kanálů, jsou zde patrné odeslané Byte Array transakce, z nichž pro každý kanál bylo dále vygenerováno určité množství LII transakcí obsahujících chybu v CRC. Dále se zde nachází informace o měření latence a propustnosti. Z hlediska těchto hodnot jsou zde 3 údaje, průměrná, maximální a minimální. Tyto hodnoty neodpovídají naměřeným hodnotám v grafech níže, protože toto je záznam ze souhrnné verifikace, kde nebyla uplatněna plná propustnost. Proto je zde minimální propustnost a latence tak nízká. Hodnotám, na které byla aplikována sekvence určená přímo pro měření, kde je nastavená plná propustnost a šířka dat je v rozmezí od 60 do 1500 bytů, budou věnovány další dvě sekce.

Následně se verifikace zaměřila na testování zmíněných registrů. Byly provedeny zápisy do důležitých registrů, většina z nich proběhla úspěšně až na zápis do resetovacích registrů, zejména se jednalo o reset komponenty PCS, který po jeho vyvolání



Obr. 6.25: Souhrn výsledků verifikace ETH PHY

způsobil, že spadla linka a už se nikdy nenahodila. Problém byl ve špatně definovaném gearboxu na RX straně, který se měl o tuto skutečnost starat. Po opravení chyby vývojářem byl obvod znovu otestován a chyba se již znovu neprojevovala.



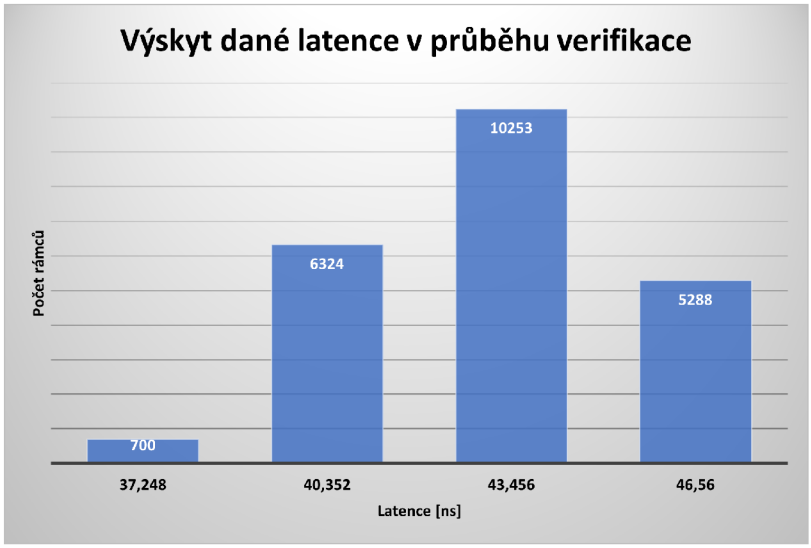
Obr. 6.26: Code coverage ETH PHY

Druhým důležitým parametrem je pokrytí komponenty ETH PHY. Jak je patrné z obrázku 6.26, dosáhlo hodnoty 77,19 %. Nižší pokrytí je způsobené tím, že nebyla prováděna zevrubná verifikace, jak již bylo zmíněno v sekci 6.2. Místo toho byla pozornost zaměřena na celkovou funkčnost návrhu. Jednalo se zejména o ověření správnosti dat na výstupu, kontroly výstupního CRC, ověření funkce pro vkládání vadného CRC, zápis do registrů pomocí MI sběrnice a na závěr měření latence

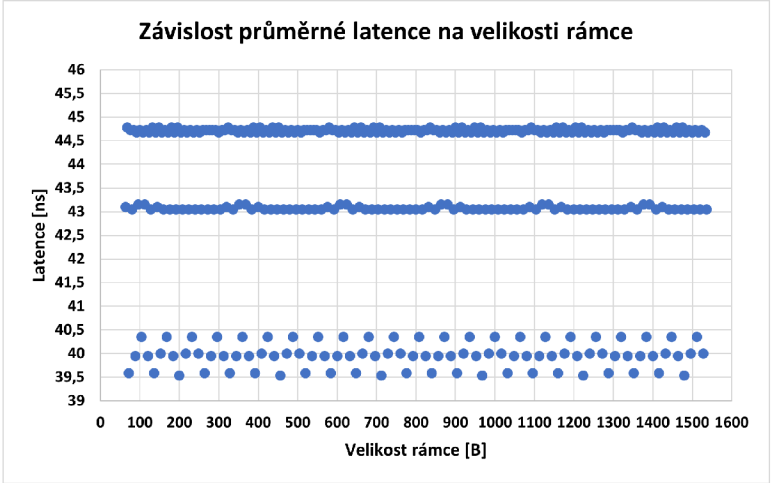
a propustnosti. Dále se na hodnotě pokrytí podílí fakt, že komponenta, která se stará o management, řídí i registry, které nejsou touto fyzickou vrstvou Ethernetu využívány.

Měření latence

Tato sekce je věnována hodnotám naměřené latence. Na obr. 6.27 je vyobrazen histogram popisující, s jakou četností se vyskytovala daná hodnota latence na kanálu 0 v průběhu celé verifikace. Jsou zde zobrazeny 4 hodnoty. Nejčetnější byla hodnota



Obr. 6.27: Graf Velikosti latence pro daný počet rámců



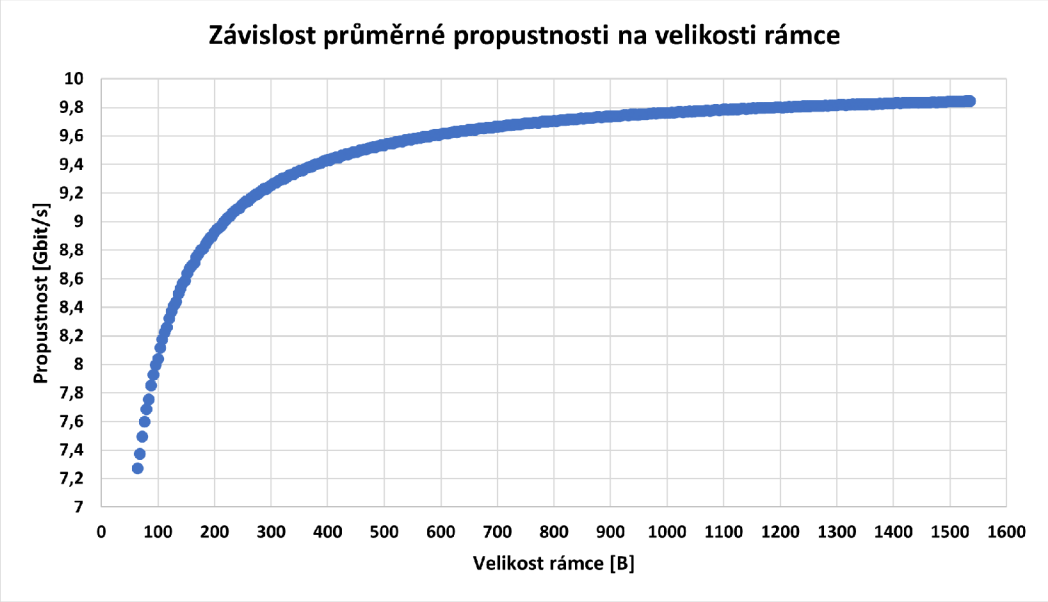
Obr. 6.28: Graf závislosti latence na velikosti rámce

43.456 ns, která se byla naměřena pro 10253 rámců. Další hodnotou je 40.352 ns, jež se nachází na druhé příčce a byla zjištěna u 6324 rámců. Dále je zde patrná maximální hodnota 46.56 ns, kterou disponovalo 5288 rámců, a na závěr nejnižší hodnota 37.248 ns, která se vyskytla na 700 rámcích.

Dále je zde vyobrazen graf závislosti průměrné latence na velikosti rámce, který je patrný z obrázku 6.28. Z těchto hodnot je patrné, že z většiny času verifikace se průměrná hodnota latence držela rámcově okolo 43 až 45 ns, což je pro účely tohoto návrhu dostačující. Dále lze s jistotou říci, že latence tohoto návrhu není závislá na délkách jednotlivých rámců.

Měření propustnosti

Druhým důležitým údajem je propustnost tohoto systému, které bude věnována tato sekce. Na obrázku 6.29 je zobrazen graf závislosti průměrné propustnosti v Gbit/s na velikosti rámce v bytech.



Obr. 6.29: Graf závislosti průměrné propustnosti na velikosti rámce

Jak je na první pohled zřejmé, hodnota propustnosti roste se zvyšující se délkou rámce. Pohybuje se od minimální hodnoty 7.271510 Gbit/s až po maximální hodnotu propustnosti 9.843661 Gbit/s. Od jisté velikosti rámce je charakteristika grafu téměř lineární. Dalším krokem bylo porovnání naměřených hodnot s těmi teoretickými. Jak teorie říká, hodnota maximální propustnosti by měla být 10 Gbit/s, když ale vezmeme v potaz nadbytečnost preambule, indikátoru `SFD` a průměrné mezipaketové mezery a dopočítáme její hodnotu podle vztahu: $propustnost = \frac{Délka\ datové\ části}{Délka\ paketu} *$

Čistá přenosová rychlost, přiblížíme se námi naměřené hodnotě. Po dosazení do zmíněného vztahu byla tedy získána hodnota 9,842519 Gbit/s, která se liší pouze nepatrně od naměřené hodnoty 9.843661 Gbit/s.

Závěr

Cílem diplomové práce **Verifikace funkčních bloků pro FPGA** bylo prostudovat problematiku funkčních verifikací, shromáždit informace o jednotlivých verifikačních modelech a jeden vybraný důkladně popsat, prostudovat problematiku Ethernetu, vytvořit vhodné verifikační prostředí ve vybrané metodologii a na závěr důkladně otestovat nízkolatenční fyzickou vrstvu Ethernetu.

V jednotlivých kapitolách teoretického úvodu byla popsána problematika verifikačních technik, zejména Formální a Funkční verifikace, a proces jejich vzniku. Dále byly porovnány dostupné modely pro vytváření verifikačních prostředí, zejména VMM, OVM, UVM a cocoTB, z nichž byli vybrány dva, UVM a cocoTB. Na základě porovnání těchto metodologií byl učiněn závěr, že nejvhodnějším kandidátem je model UVM. V další řadě byla tedy nastíněna problematika UVM. Jmenovitě se jednalo o popis následujících částí: hierarchie tříd, fázovací systém, Transaction Level Modeling (TLM) a funkcionalita jednotlivých komponent metodologie UVM.

Další částí této práce bylo studium Ethernetu. V tomto odvětví se práce zaměřuje zejména na objasnění teorie pohybující se okolo **10GBASE-R** Ethernetu, jehož problematika je rozdělena na popis Spojové a Fyzické vrstvy. První část nejprve vytváří obecný pohled na problematiku a následně popisuje jednotlivé podvrstvy Logical Link Control (LLC) a Media Access Control (MAC). Jelikož verifikovaný návrh neobsahuje LLC vrstvu, tak je důkladně popsána MAC vrstva, zejména popis formátu rámce a procesy vysílání a příjmu. Druhá část se nejprve zaměřuje na obecný popis problematiky Fyzické vrstvy a následně svou pozornost přesouvá na podvrstvu PCS, kde je podrobně popsána její struktura, zejména bloky Encode, Scramble, Gearbox, Block sync, Descramble a Decode. V závěru teoretické části jsou povrchově popsány vrstvy PMA a PMD a rozhraní XGMII, kterým není v práci věnována přílišná pozornost.

V praktické části byla v prvé řadě prostudována specifikace verifikovaného obvodu. Následovalo vytvoření prostředí modelu UVM, jehož funkcionalita byla popsána v kapitole 5. Jmenovitě se zde nachází popis funkcionality proprietárního LII rozhraní, následně dalších prostředí, které bylo potřeba vytvořit k ověření funkčnosti daných komponent a jejich virtuálních rozhraní.

Dalším krokem byla samotná verifikace. Nejprve se práce zaměřuje na verifikaci dílčích komponent, díky čemuž jsou odhaleny chyby vzniklé neošetřením okrajových podmínek. Jmenovitě se jedná o problém s nastavením validních bytů při shození RDY signálu a špatného vyhodnocování stavu linky u komponenty RX PCS. Dále pak o chybu při vyhodnocování validity CRC a reakce na sekvenční chyby u komponenty RX MAC.

Následujícím krokem je verifikace celé top komponenty ETH PHY, kde je dáván

důraz zejména na měření latence a propustnosti. Z těchto údajů je vyhodnocena průměrná latence 43,12 ns, která je pro funkci návrhu dostačující. Dále je změřena propustnost, jejíž maximální hodnota odpovídá teoretické, z čehož lze vyvodit, že návrh byl zkonstruován dle předpokladů standardu IEEE 802.3. Dalším krokem je zkoumání zápisu do jednotlivých registrů pomocí MI sběrnice. Z tohoto pozorování vyplývají chyby, které souvisí se zápisem do těchto registrů, zejména se jedná o resetování dílčích komponent.

V poslední fázi se verifikace zaobírá vylepšením Code coverage zmíněných komponent, jehož hodnota nebyla přijatelná. Zlepšení bylo docíleno vytvořením nových sekvencí, které byly popisovány v tabulkách 5.2, 5.3 a 5.1. Díky zvětšení knihovny sekvencí se podařilo zvýšit pokrytí na přijatelné hodnoty, jež jsou patrné z obrázku 6.1. Nižší pokrytí komponent je způsobeno nepokrytím zejména nepoužitých řádku kódu a podmínkami, které nemají správně nikdy nastat. Podrobněji jsou důvody popsané v sekcích, které popisují verifikace příslušných komponent.

Komponenta	Code coverage [%]
TX MAC	95,36
TX PCS	81
RX PCS	92,66
RX MAC	94,47
ETH PHY	77,19

Tab. 6.1: Tabulka Code coverage

Pokračováním diplomové práce by mohlo být otestování sériové linky vedoucí mezi RX a TX stranou, čímž by se mohla důkladněji prověřit funkcionalita gearboxů. Dále pak vytvoření formální verifikace, která je zevrubnější a mohla by odhalit další možné chyby. Z předchozích informací lze vyvodit skutečnost, že k důkladnému otestování nestačí jen pouhá simulace, ale je zapotřebí návrh podrobit verifikací, která umožní objevit chyby, které se neprojeví v simulaci, ale ani v reálném testování.

Literatura

[1] Formal verification. 2019. Techdesignforums [online]. [cit. 2021 20-10] Dostupné z: <https://cutt.ly/qYmn3OE>

[2] WANG, Farn. Theorem proving: Formal Methods [online]. [cit. 2021 20-10] Electrical Engineering National Taiwan University. Dostupné z: <https://cutt.ly/aYmmIKe>

[3] KATZ, David. 2012. Using formal methods for sophisticated static code analysis. Embedded [online]. [cit. 2021 22-10] Dostupné z: <https://cutt.ly/qYmmGU7>

[4] BERGERON, Janick, Eduard CERNY, Alan HUNTER and Andrew NIGHTINGALE. 2005. Verification Methodology Manual for SystemVerilog.

[5] VERIFICATION PLAN. 2017. Testbench [online]. testbench. [cit. 2021 23-10] Dostupné z: <http://www.testbench.in/TS_24_VERIFICATION_PLAN.html>

[6] COVERAGE DRIVEN CONSTRAINT RANDOM VERIFICATION ARCHITECTURE. 2017. Testbench [online]. testbench. [cit. 2021 26-10] Dostupné z: <https://cutt.ly/sYmm6v0>

[7] Code Coverage. Semiengineering [online]. [cit. 2021 26-10] Dostupné z: <https://cutt.ly/JYmQgoN>

[8] KUMAR TALA, Deepak. 2014. Random Constraints Part-I. Asic-world [online]. [cit. 2021 26-10] Dostupné z: <http://www.asic-world.com/systemverilog/random_constraint1.html>

[9] Assertion-based verification. 2019. Techdesignforums [online]. techdesignforums. [cit. 2021 26-10] Dostupné z: <https://www.techdesignforums.com/practice/guides/guide-to-assertion-based-verification/>

[10] KUMAR TALA, Deepak. 2014. Functional Coverage Part-I. Asic-world [online]. [cit. 2021 26-10] Dostupné z: <http://www.asic-world.com/systemverilog/coverage1.html>

[11] Verification Methodologies. Semiengineering [online]. semiengineering. [cit. 2021 28-10] Dostupné z: <https://semiengineering.com/knowledge_centers/eda-design/verification/methodology/>

[12] Open Verification Methodology. Cadence [online]. cadence. [cit. 2021 28-10] Dostupné z: <https://cutt.ly/pYmQmc3>

[13] UVM, OVM and VMM. 2019. Aldec [online]. aldec. [cit. 2021 28-10] Dostupné z: <https://cutt.ly/NYmQORV>

[14] WHY COCOTB ? 2021. Fpga-tools [online]. Dostupné z: <https://cutt.ly/pYmQJIo>

[15] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009), Feb 2013.

[16] Uvm-tutorial. 2021. Chipverify [online]. chipverify. [cit. 2021 30-10] Dostupné z: <https://www.chipverify.com/uvm/uvm-tutorial>

[17] Universal Verification Methodology (UVM) 1.2 Class Reference. 2014. 1370 Trancas Street 163, Napa, CA 94558, USA.: Accellera Systems Initiative.

[18] SINGHAL, Manish. 2016. UVM Phasing. Learnuvmverification [online]. learnuvmverification. [cit. 2021 01-11] Dostupné z: <https://cutt.ly/sYmQ0A6>

[19] TLM1 Interfaces, Ports, Exports and Transport Interfaces. Verificationacademy [online]. [cit. 2021 02-11] Dostupné z: <https://cutt.ly/cYmQ75V>

[20] SPURGEON, Charles E and Joann ZIMMERMAN. 2014. Ethernet: The Definitive Guide. Vyd. 2. Boston: O´Reilly.

[21] WESLEY, Chai, Alissa IREI and John BURKE. 2020. Ethernet. Techtarget [online]. Dostupné z: <https://www.techtarget.com/searchnetworking/definition/Ethernet>

[22] Data-link Layer Introduction [online]. Dostupné z: <https://cutt.ly/bYmWrYX>

[23] Highteck.net. *Ethernet* [online]. [cit. 2021-11-06]. ISBN 978-1-5044-5090-4. Dostupné z: <http://www.highteck.net/EN/Ethernet/Ethernet.html>.

[24] HAMMAD, madhuri. 2021. Logical Link Control (LLC) Protocol Data Unit. Geeksforgeeks [online]. Dostupné z: <https://cutt.ly/HYmWvNG>

[25] Medium Access Control (MAC). 2020. Electronics Research Group [online]. Dostupné z: <https://erg.abdn.ac.uk/users/gorry/course/lan-pages/mac.html>

[26] IEEE COMPUTER SOCIETY. *IEEE Standard for Ethernet: Section 1* [online]. IEEE Std 802.3™-2018 (Revision of IEEE Std 802.3-2015). New York

(USA), 2018 [cit. 2021-11-14]. ISBN 978-1-5044-5090-4. Dostupné z: <https://ieeexplore.ieee.org/servlet/opac?punumber=8457467>

[27] IEEE COMPUTER SOCIETY. *IEEE Standard for Ethernet: Section 4* [online]. IEEE Std 802.3™-2018 (Revision of IEEE Std 802.3-2015). New York (USA), 2018 [cit. 2021-11-14]. ISBN 978-1-5044-5090-4. Dostupné z: <https://ieeexplore.ieee.org/servlet/opac?punumber=8457467>

[28] Tutorial - Using Modelsim for Simulation, for Beginners. Nandland [online]. Dostupné z: <https://cutt.ly/iYmWTQD>

Seznam symbolů a zkratek

API	Application Programming Interface
BER	Bit Error Rate
CD	Collision Detection
CocoTB	COroutine based COsimulation TestBench
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DIC	Deficit Idle Count
DPI	Direct Programming Interface
DSAP	Destination Service Access Point
DUT	Design Under Test
EDA	Electronic Design Automation
FCS	Frame Check Sequence
FIFO	First In First Out
FPGA	Field Programmable Array
GTY	Gigabit transceivers
GUI	Grafical User Interface
ID	Identification
IP	Internet Protocol
LAN	Local Area Network
LII	Low latency Independent Interface
LLC	Logical Link Control
MAC	Medium Access Control
NIC	Network Interface Controller
OSI	Open Systems Interconnection

OVL	Open Verification Library
OVM	Open Verification Library
PCS	Physical Coding Sublayer
PDU	Protocol Data Unit
PHY	Physical layer
PMA	Physical Medium Attachment
PMD	Physical Medium Dependent
REJ	Rejected
RNR	Receive Not Ready
RR	Receive Ready
RS	Reconciliation Sublayer
RX	Receive
SFD	Start Frame Delimiter
SSAP	Source Service Access Point
SVA	System Verilog Assertion
TLM	Transaction Level Modeling
TX	Transmit
UVM	Open Verification Library
VHDL	VHSIC Hardware Description Language
VMM	Open Verification Library
WAN	Wide Area Network
XGMII	10 Gigabit Media Independent Interface