VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PROSTŘEDÍ PRO FUNKČNÍ VERIFIKACI MULTI-SBĚRNIC PODLE UVM STANDARDU

ENVIRONMENT FOR FUNCTIONAL VERIFICATION

OF MULTI-BUSSES BASED ON UVM STANDARD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ BENEŠ
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student:	**Beneš Tomáš**
Program:	Informační technologie
Název:	**Prostředí pro funkční verifikaci multi-sběrnic podle UVM standardu** **Functional Verification Framework for Multi Buses Following the UVM Standard**
Kategorie:	Vestavěné systémy

Zadání:

1. Prostudujte oblast funkční verifikace číslicových obvodů a zejména verifikační metodologii UVM.
2. Seznamte se s návrhem a fungováním multi-sběrnic pro zpracování vícero datových paketů za jeden hodinový cyklus na FPGA čipech.
3. Navrhněte soubor základních verifikačních komponent podle UVM standardu pro testování modulů pracujících na multi-sběrnicích.
4. Implementujte navržené UVM komponenty v jazyce SystemVerilog.
5. Zvolte alespoň tři firmwarové moduly a vytvořte zapojení pro jejich funkční verifikaci s využitím implementovaných UVM komponent. Vhodné firmwarové moduly dodá vedoucí práce.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.
- L. Kekely, J. Cabal, V. Puš and J. Kořenek, "Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs," 2020 23rd Euromicro Conference on Digital System Design (DSD), 2020, pp. 49-56, doi: 10.1109/DSD51259.2020.00020. https://ieeexplore.ieee.org/abstract/document/9217811

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Kekely Lukáš, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	11. května 2022
Datum schválení:	29. října 2021

Abstrakt
Práce se zabývá návrhem a následnou implementací prostředí pro verifikace multi-sběrnic s využitím principů univerzální verifikační metodologie (UVM). Dále se zabývá implementací verifikací tří FPGA konkrétních komponent využívající multi-sběrnice jako vstupní a výstupní rozhraní. Implementace prostředí i všech verifikací je napsaná v jazyce SystemVerilog s využitím knihovny implementující základní konstrukce pro UVM. Dosažené výsledky práce jsou funkční a jednoduše znovupoužitelné při tvorbě dalších verifikací využívající multi-sběrnic. Navržené prostředí se dají využít jako struktura pro tvorbu dalších verifikačních prostředí pro jiné sběrnice.

Abstract
This thesis focus on the design and subsequent implementation of a multi-bus verification environment using the principles of the Universal Verification Methodology (UVM). It also focus on the implementation of the verification of three FPGA components using multi-bus as input and output interfaces. The implementation of the environment and all verifications is written in SystemVerilog language using a library that implement the basic constructs for UVM. The achieved results of the work are functional and easily reusable when creating further verifications using multi-bus. The proposed environments can be used as a structure for creating other verification environments for other buses.

Klíčová slova
UVM, Verifikace, Funkční verifikace, FPGA, Multi-sběrnice

Keywords
UVM, Verification, Functional verification, FPGA, Multi buses

Citace
BENEŠ, Tomáš. *Prostředí pro funkční verifikaci multi-sběrnic podle UVM standardu.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Lukáš Kekely, Ph.D.

Prostředí pro funkční verifikaci multi-sběrnic podle UVM standardu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Kekelyho, Ph. D. Další informace mi poskytl můj kolega Ing. Radek Iša. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Tomáš Beneš
11. května 2022

Poděkování

Rád bych poděkoval vedoucímu mé práce Ing. Lukášovi Kekelymu, Ph.D. za cenné rady a připomínky, které mi značně pomohly v rámci tvorby této práce. Také bych rád poděkoval všem kolegům ze sdružení CESNET za spolupráci a rady při implementaci. Moje poděkování patří i Ing. Radku Išovi za cenné rady a spolupráci v rámci návrhu celého prostředí.

Obsah

1 Úvod **2**

2 Verifikace číslicových obvodů **3**
2.1 Funkcionální verifikace 4
2.1.1 Funkcionální pokrytí 5
2.1.2 Pokrytí kódu 5
2.2 Verifikační plán 6
2.3 Implementace verifikací 7

3 UVM **9**

4 Multi-sběrnice **14**
4.1 Specifikace MFB 14
4.2 Specifikace MVB 16

5 Tvorba verifikačních prostředí **18**
5.1 Prostředí logic_vector 18
5.2 Prostředí byte_array 19
5.3 Multi Frame Bus (MFB) 19
5.4 Multi Value Bus (MVB) 21

6 Aplikace prostředí na tvorbu konkrétních verifikací **23**
6.1 Komponenta asfifox 23
6.1.1 Verifikační plán 25
6.1.2 Implementace 26
6.2 Komponenta MFB rozdělovač 28
6.2.1 Verifikační plán 29
6.2.2 Implementace 30
6.3 Komponenta metadata extraktor 32
6.3.1 Verifikační plán 33
6.3.2 Implementace 34

7 Vyhodnocení výsledků **36**
7.1 Možné rozšíření systému 36

8 Závěr **38**

Literatura **39**

Kapitola 1

Úvod

V počítačových sítích se v dnešní době přenáší stále více dat, která jsou stále objemnější. Z tohoto důvodu je potřeba zrychlovat zpracování síťové komunikace na co nejnižší úrovni. I přesto, že jsou dnešní procesory stále rychlejší, tak nastává problém s propustností. Každý procesor může v jednu chvíli zpracovávat pouze jeden požadavek. Zde narážíme na omezení klasických procesorů. Pro dosažení dostatečné propustnosti by se procesory musely zrychlit anebo by jich muselo být velké množství. Tato řešení jsou ale velice finančně a časově náročná! Řešením tohoto problému může být použití hardwarové akcelerace s pomocí speciálních rozhraní podporujících přenos a následné zpracování více různých hodnot v jednom hodinovém taktu.

I když nám hardwarová akcelerace dovoluje zpracovávat požadavky paralelně, tak má i nějaké nevýhody. Jednou z největších nevýhod je odhalování chyb. Při samotné implementaci je odhalení chyby velice těžké a v některých případech skoro nemožné. Zde přicházejí na scénu verifikace. Je potřeba vytvořit testy, které dokáží odhalit nepřesnosti mezi specifikací a reálně implementovanou komponentou. Pokud by se verifikace neintegrovaly do vývojového procesu, tak by mohlo docházet k nekorektnímu chování přímo na nasazeném zařízení. To může mít v některých případech až katastrofické důsledky.

Z těchto důvodů se v bakalářské práci zaměřuji na vytvoření prostředí pro funkční verifikaci multi-sběrnic pomocí v této oblasti rozšířeného standardu UVM. Na implementaci je použit jazyk SystemVerilog a knihovna implementující základní konstrukce. Prostředí i konkrétní verifikace jsou navrženy s ohledem na spolupráci se sdružením CESNET.

Práce je složená z osmi kapitol. V úvodu seznamuji s tématem. V kapitole č. 2 popisuji rozdělení a problematiku verifikací číslicových systémů. Následně v kapitole č. 3 představuji koncept UVM, vysvětluji proč je samotný UVM standard důležitý. V kapitole č. 4 představuji samotnou problematiku multi-sběrnic, a také popisuji specifikaci jednotlivých sběrnic. V kapitole č. 5 vytvářím koncept systému využívající UVM pro tyto sběrnice. Také zde uvádím strukturu, propojení a postup tvorby prostředí. V kapitole č. 6 popisuji specifikaci, průběh tvorby a aplikaci UVM verifikačních prostředí na tvorbu tří verifikací pro reálné FPGA komponenty. V kapitole č. 7 uvádím výsledky aplikace prostředí a postupů na verifikace. Také popisuji samotné dosažené výsledky všech verifikací a možná rozšíření. V poslední kapitole 8 shrnuji postup a vyhodnocuji výsledky práce.

Kapitola 2

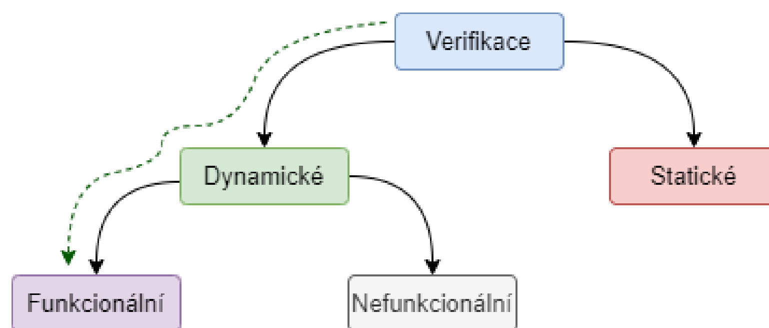
Verifikace číslicových obvodů

Verifikace [9] je obecně proces ověřování nejen správnosti systému, přístroje, ale i výroků a hypotéz, pomocí aplikace testů, či pomocí formální analýzy. Tento proces dokáže odhalit chyby a nedostatky entity, kterou verifikujeme. Nutnost odhalovat a opravovat chyby je hlavní důvod, proč by se měly verifikace tvořit. A to zvláště u produktů, které mohou mít katastrofální důsledky při chybě. Jako jeden ze známých příkladů mohu uvést let Apolla 13, kde u termostatického spínače pro ohřívače kyslíkových nádrží bylo změněno provozní napětí z 28 na 65V [11]. Bohužel už nebyla změněna specifikace napětí a to ve výsledku znamenalo výbuch a neúspěch celé mise. Naštěstí se nikomu nic nestalo a všichni přežili. Jednalo se tedy o chybu správy konfigurace. A tato chyba mohla být odhalena právě pomocí verifikace.

Tvorba verifikací je důležitá i při vývoji FPGA projektů. Vývoj FPGA komponent je značně komplexní a je jednoduché při samotném vývoji udělat chyby, nebo neošetřit veškeré případy užití. Při nezařazení verifikací do procesu vývoje je pak velice jednoduché zanechat chyby v konečném produktu. To by mohlo mít za důsledek zhoršení pověsti vývojářů a znehodnocení celé jejich práce. Proto je velmi podstatné nasazovat verifikace do procesu vývoje FPGA projektů.

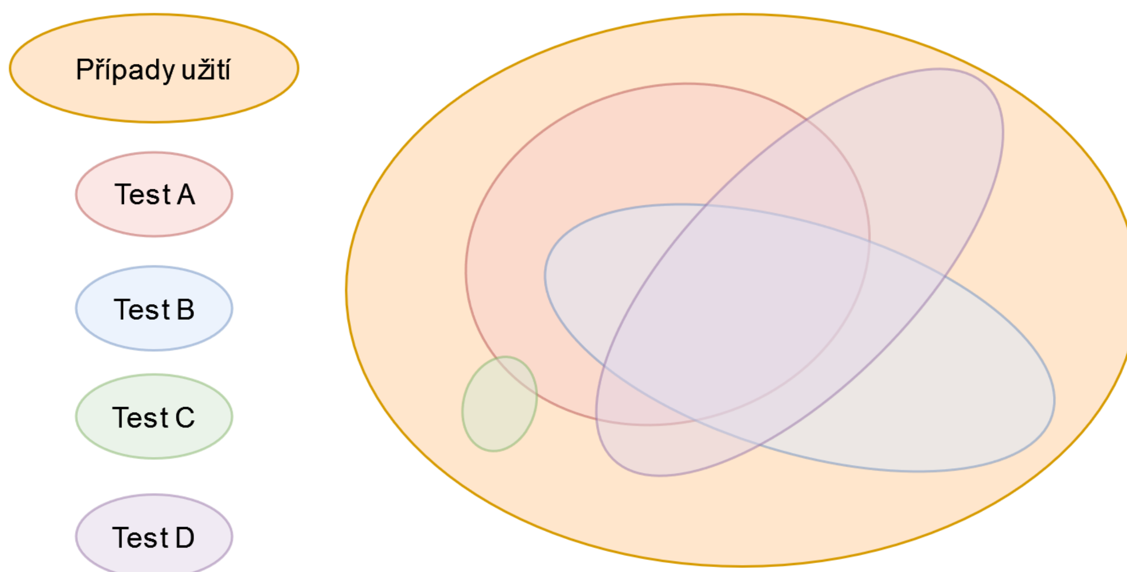
Tendence a nutnost vytvářet verifikace potvrzují i některé studie [10]. Podle nich průměrný čas FPGA projektu v roce 2020 strávený na verifikaci je 51 %. Což je od roku 2014 nárůst o 5 %. Také se od roku 2014 zvýšil průměrný počet vývojářů pracujících na verifikacích o 5.5 %. Oproti tomu zvýšení průměrného počtu vývojářů pracujících na samotném designu vzrostl o pouhé 1.5 %.

Verifikace se mohou dělit na několik typů. Rozdělení verifikací je graficky znázorněno na obrázku č. 2.1. V rámci této práce je pro nás nejzajímavější pouze jeden typ. Na obrázku je tento typ vyznačen zelenou šipkou. Můžete si všimnout, že první rozdělení je na statické a dynamické verifikace. Statické verifikace jsou založeny na manuální kontrole samotného kódu. Patří sem například formální verifikace, dodržování stanovených konvencí (používají se v kódu běžně používané konstrukce, má kód správnou formu atd.) a mnoho dalších. V souvislosti s tvorbou funkčních verifikací jsou ale více zajímavé dynamické verifikace. Při tomto přístupu se nad komponentami spouští řada testů a kontroluje se správnost chování. Testy jsou vytvořeny tak, aby pokryly co nejvíce případů užití produktu. Některé testy jsou vytvořeny na otestovaní nejčastěji dosahovaných případů. Ale většina testů je tvořena za účelem otestovat některé krajní případy.



Obrázek 2.1: Rozdělení verifikací podle jejich typu

Pokrytí případů užití je znázorněno na obrázku č. 2.2. Tento obrázek zachycuje veškeré případy užití jako množinu. Testy jsou znázorněny jako množina pokrytí případů užití. Některé testy mohou testovat stejnou funkcionalitu, ale je žádoucí, aby pokrývaly i nové případy. Z reálného hlediska je množina případů užití tak velká, že nikdy nedokážeme otestovat 100 % případů. Proto je důležité otestovat hlavně nejvíce zřejmé krajní případy.



Obrázek 2.2: Znázornění otestování případů užití

Z obrázku č. 2.1 můžeme vyčíst ještě další možné rozdělení dynamické verifikace na funkcionální a nefunkcionální. Nefunkcionální verifikace se, jak již název napovídá, nezaměřují na funkcionalitu, ale spíš na ostatní aspekty provozu komponenty. Jedná se většinou o metriky komponenty. To může být například jak dlouho trvá produktu vykonat nějakou akci, počet výpadků systému za určitou časovou jednotku, spolehlivost uchování dat a mnoho dalších. Pro tuto práci je zásadní spíše funkcionální verifikace.

2.1 Funkcionální verifikace

Tento přístup [9] k verifikacím se snaží ověřit co nejvíce funkcionalit komponenty. Ze specifikace komponenty se vymodeluje zjednodušený model, pomocí kterého následně budeme

ověřovat správné chování původní komponenty. Pro ověření funkcionalit se navrhují testovací vektory, které se aplikují na testovanou komponentu. Testovací vektory jsou předány také do modelu. Za předpokladu, že je model správně namodelován, můžeme předpokládat, že se chová komponenta korektně, pokud je chování komponenty a modelu totožné. U tohoto typu verifikací máme několik druhů metrik, které nám určují, jak dobře je funkcionální verifikace vytvořena a v jakém stádiu se aktuálně nachází. Metriky se dají základně rozdělit na automaticky generované jako například pokrytí kódu. A funkcionální pokrytí, což je specifická metrika pro každou verifikaci.

2.1.1 Funkcionální pokrytí

Funkcionální pokrytí je definováno přímo vývojářem. Pomocí tohoto pokrytí se dá ověřit to, že jsou některé z funkcionalit komponenty správně implementovány. Dále toto pokrytí dělíme na tři typy.

Orientováno na data - Kontroluje, jaká data byla do komponenty poslána. Také může kontrolovat kombinace dat na více vstupních/výstupních rozhraních. Tento typ pokrytí je většinou dosti jednoduchý na implementaci.

Orientováno na chování - Sleduje, jestli se objevily určité sekvence chování. Například zaplnění vstupní/výstupní linky komponenty.

Pokrytí tvrzení - Tento druh pokrytí kontroluje různá tvrzení. Může kontrolovat to, že nenastane nějaký nežádaný jev (například čtení a zápis najednou). Také může kontrolovat, že se nějaká událost stala alespoň jednou za verifikační běh (například zaplnění vstupního zásobníku).

Ve funkcionální verifikaci by se také měla určit sada funkcí komponenty, u které by se měla provádět kontrola. Následovným testováním pomocí dříve uvedených postupů a analýzou funkcionalit se dají odhalit nedostatky verifikace. Pomocí těchto nedostatků můžeme upravit verifikaci a napsat nové testy.

2.1.2 Pokrytí kódu

Další metrikou je pokrytí kódu. Tato metrika udává, jaký kód byl vykonán. A s jakými hodnotami proměnných byl vykonán. Pomocí tohoto pokrytí můžeme najít nedostatky ve verifikacích. Můžeme pomocí této metriky zjistit, jaké části kódu se vůbec nevykonávají. Díky této informaci víme, jaké testy jsou nutné přidat, aby byla tato část kódu vykonána. To, že má verifikace 100% pokrytí kódu bohužel neznamená, že komponenta provádí to, co chceme. K pokrytí kódu se dá přistoupit následujícími způsoby:

Pokrytí řádků - Toto pokrytí zjišťuje, jestli se provedly všechny řádky kódu komponenty.

Pokrytí větvení - Toto pokrytí zjišťuje, jestli se prošly veškeré větve programu komponenty.

Pokrytí stavů podmínek - Toto pokrytí zjišťuje, jestli se provedly veškeré kombinace logických operací určité podmínky.

Pokrytí výrazů - Toto pokrytí funguje stejně jako pokrytí stavů podmínek, ale aplikuje se na veškeré výrazy v kódu.

Pokrytí stavů konečného automatu - Toto pokrytí zjišťuje, jestli byly veškeré stavy konečného automatu aktivní. A také jestli byly vykonány veškeré přechody mezi stavy.

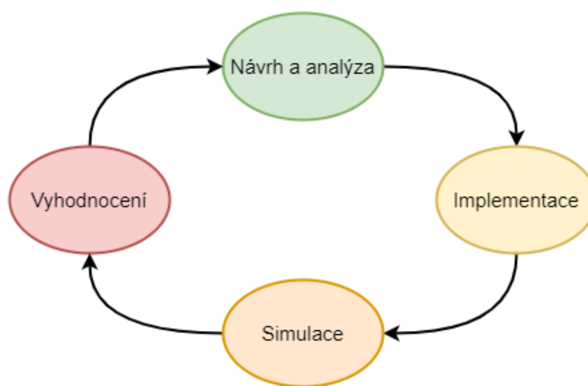
Pokrytí přepnutí stavů - Toto pokrytí zjišťuje, jestli veškeré proměnné změnily svůj stav do veškerých možných hodnot.

2.2 Verifikační plán

Jelikož jsou verifikace iterativní proces, tak je důležité si udržovat informace o tom, v jakém stavu se aktuálně nachází. Také je produktivní si před implementací samotné verifikace nastavit cíle, kterých chceme verifikací dosáhnout. Tyto cíle si vytvoříme pomocí dříve uvedených metrik v sekcích 2.1.1 a 2.1.2. Pro tyto účely je dobré si připravit verifikační plán, který nám pomůže dosáhnout všech nastavených cílů. Vývoj komponenty může probíhat současně s její verifikací, a tak se v průběhu implementačního procesu může plán verifikace měnit a rozšiřovat.

Na obrázku č. 2.3 je naznačen průběh tvorby verifikace. Na začátku procesu se analyzuje komponenta a navrhnou se cíle, kterých chceme dosáhnout. Následně se pokusíme implementovat testy na dosažení navržených cílů ve verifikaci. Po implementaci spustíme testy a vytvoříme si zprávy zobrazující metriky. Z vytvořených zpráv následně vyhodnotíme dosažení navržených cílů. Verifikaci prohlásíme za hotovou, pokud jsme dosáhli dostatečného počtu a úrovně pokrytých cílů a pokud se nezměnila implementace, nebo specifikace komponenty. Verifikační proces začneme od začátku, pokud některá z podmínek ukončení nevyhovuje.

Proces aplikace testů a stimulů na komponentu se nazývá simulace. Pro simulování VHDL komponent existuje spoustu simulačních nástrojů s velkým rozsahem funkcionalit. V rámci této práce jsem použil simulační nástroj ModelSim, který obsahuje nástroje pro jednoduché generování zpráv o pokrytí kódu. V této zprávě jsou přehledně dostupné záznamy o tom, jaké je pokrytí kódu v celé komponentě. Také je zde uvedeno pokrytí kódu v jejích podkomponentách. Pro každou zprávu se dá zobrazit pokrytí kódu pomocí jednotlivých přístupů k pokrytí kódu (viz 2.1.2).



Obrázek 2.3: Průběh tvorby verifikace

Verifikační plán by měl obsahovat výstižné informace o cíli, kterého chceme dosáhnout. Z uvedených informací musí být na první pohled zřejmé, co jimi bylo myšleno i po delší době. Informace uvedené v plánu mohou být složeny z políček podle preferencí vývojáře. Plán může být složen například z políček uvedených v následujícím seznamu.

- Název testu
- Sekce - k čemu se daný cíl vztahuje. Například vstupní/výstupní rozhraní, vnitřní funkcionalita atd.
- Popis - bližší popis testu
- Způsob pokrytí - pokrytí kódu, pokrytí určité funkcionality
- Priorita - čím vyšší priorita, tím dříve by se měla funkcionalita otestovat

Výhodou verifikačního plánu je, že se do něj dají postupně přidávat další cíle verifikace. Také je výhodou, že máme všechny cíle zapsané na jednom místě, takže bychom v průběhu implementace na žádný neměli zapomenout. Podle mého názoru se tvorba verifikačního plánu vyplatí u velkých komponent, kde je větší pravděpodobnost rozšíření v průběhu implementace. Menší komponenty se dají otestovat v podstatě v jedné iteraci verifikačního procesu a tak by byla tvorba plánu zbytečné plýtvání časem. Existují hlavní dva přístupy k tvoření verifikačního plánu, jimiž jsou rozbor zevnitř ven a rozbor z venku dovnitř.

Prvním přístupem je rozbor zevnitř ven, který je orientovaný na design. Tento přístup analyzuje komponentu po blocích, případně po rozhraních. Je pro něj potřebný detailnější návrh a specifikace implementace. Proto se tento přístup nehodí u komponent, které ještě nejsou implementované anebo jsou v rané fázi implementace. Může vést k velkému množství cílů, které by byly nemožné splnit v rozumném čase. A jelikož je přístup hodně nízkoúrovňový, tak může dojít k pokrytí, které ale vůbec nemusí vypovídat o funkčnosti komponenty. Tento přístup se hodí spíše pro menší komponentu s dobrou specifikací implementace. Je výhodné přístup použít i u komponent, které se budou používat v mnoha různých nasazeních s různým přístupem k práci s komponentou.

Druhý přístup je zaměřený spíše na komponenty, u kterých víme, jak se s nimi bude pracovat. Je dobré u těchto komponent mít přístupnou jak specifikaci komponenty, tak i specifikaci případů užití komponenty. Tento přístup nám může ukázat zajímavější informace o pokrytí, jako například nedokončené a nefunkční funkcionality komponenty. Tento přístup se hodí tedy spíše na větší komponenty, u kterých by byl přístup rozboru zevnitř ven velmi rozsáhlý. Také je vhodný spíše u komponent, které se použijí pouze v omezeném rozsahu použití a je jasné, jakým způsobem se s nimi bude pracovat.

Z výše uvedených možností si myslím, že by byla nejlepší kombinace obou přístupů. Nejdříve navrhnout cíle pomocí rozboru z venku dovnitř, které nepotřebují specifikaci implementace. Z tohoto přístupu dostaneme základní povědomí o funkcionalitách, které můžeme ověřit. Následně při dostatečné implementaci komponenty můžeme použít rozbor zevnitř ven. Ten nám přidá cíle zaměřené spíše na data a pokrytí rozhraní.

2.3 Implementace verifikací

Pomocí výše uvedených postupů a technik dokážeme navrhnout plán, jakým se bude verifikace implementovat. Také díky nim máme dostupné metriky, které nám udávají aktuální postup ve vývoji. A můžeme se podle nich rozhodnou, jestli je verifikace dostatečně komplexní. Dokonce máme i nástroje pro průběžnou údržbu. Teď stačí pouze verifikaci naprogramovat a otestovat funkcionality. Zde nám nastává problém. Existuje mnoho různých přístupů k tvorbě verifikací, kde některé jsou již zastaralé. Můžeme si samozřejmě zvolit vlastní metodologii pro tvorbu verifikací. To je ovšem časově náročné na vymyšlení a následnou implementaci konstrukcí, které budou ulehčovat implementaci samotných verifikací.

Velkým rizikem je také rozšířitelnost a udržitelnost při špatném návrhu metodologie. Toto riziko je zřetelné i z historického pohledu. Existuje mnoho druhů již vytvořených metodologií, kterými jsou například RVM, AVM, URM a OVM [6]. Tyto metodologie jsou poměrně rozdílné a každá měla svoji komunitu vývojářů, kteří je používali, nebo používají. Na těchto metodologiích pracovalo mnoho lidí v rozmezí několika let. I přesto se stále objevují nové a lepší. Proto si myslím, že není vhodné vymýšlet novou metodologii pro tvorbu verifikací. Lepším řešením je adoptovat nějakou metodologii již vytvořenou a běžně používanou. Metodologie rozšířená a udržovaná má velkou výhodu při hledání chyb na internetu. Také je jednoduší najít někoho, kdo nám s naším specifickým problémem pomůže.

Naštěstí byla v roce 2011 vydána nová univerzální verifikační metodologie. UVM[2] je pravděpodobně nejrozšířenější metodologie . Má také svoji komunitu, která je ochotná podat pomocnou ruku ve formě rady na fórech. Bližší popis UVM naleznete v kapitole 3. UVM má jasně popsaný přístup k mnoha problémům, které by mohly nastat v průběhu implementace a je vcelku jednoduchá na adoptování. Z těchto a mnoha dalších důvodů je rozumné použít právě UVM.

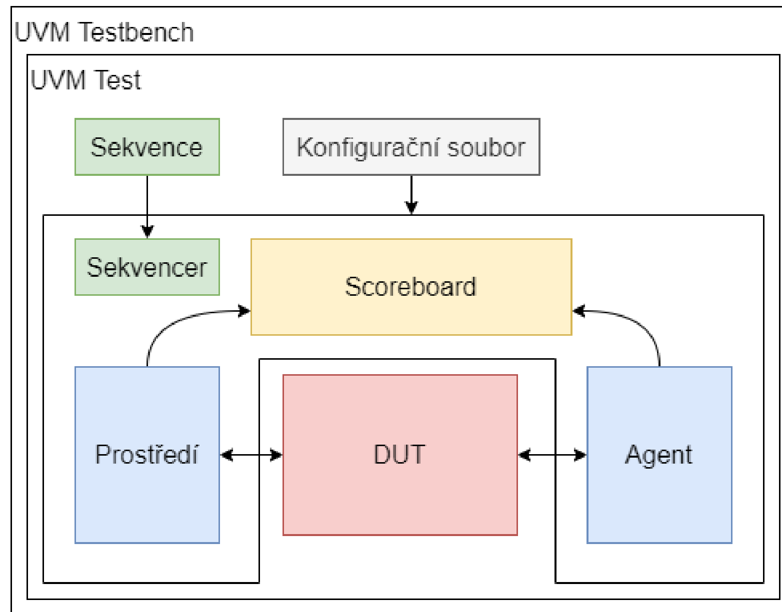
Pokud zvolíme UVM, tak je výběr programovacího jazyka vcelku triviální. Při vydání UVM byla také vytvořena knihovna základních komponent pro jazyk SystemVerilog, který by měl být v následujících 12 měsících nejvíce používaný ze všech [10]. SystemVerilog je objektově orientovaný programovací jazyk určený pro verifikaci a popis hardware [1]. Podporuje například podmíněné generování hodnot proměnných, kontrolu tvrzení, sběr dat potřebných k funkčnímu pokrytí orientovaného na data, asynchronní funkce a mnoho dalšího. Takže je pro naše použití naprosto vyhovující.

Kapitola 3

UVM

UVM [6][7][2] je standardizovaná metodologie zaměřená na tvorbu verifikací číslicových systémů. Adresuje co největší množství obecných problémů při tvorbě verifikací a navrhuje nejlepší postupy, jak tyto problémy řešit. Hlavní problémy, které UVM řeší jsou design a práce s daty, generování stimulů, struktura projektu, spouštění a řízení testů, systém pro sběr a následné hlášení zpráv, chyb a dalších užitečných druhů výpisů. Postupy v ní uvedené jsou navrženy tak, aby verifikace implementovaná pomocí této metodologie byla co nejlepší a pokrývala co největší množství případů použití verifikované komponenty. Snaží se o co největší znovupoužitelnost a jednoduchou rozšířitelnost a průběžnou údržbu. K tomu jí dopomáhá objektově orientovaný návrh. Navrhuje běžné komponenty s minimální funkcionalitou, které se dají následně rozšířit podle potřeb zrovna implementované verifikace. Využívá také návrhových vzorů pro řešení některých problémů. Například využívá továrny pro vytváření a následný přístup k objektům bez nutnosti předávání referencí skrze celou strukturu projektu.

Pomocí UVM dokáže vývojář implementovat v podstatě jakékoliv prostředí. Dosahuje toho za použití hierarchie UVM komponent, konfigurační databáze, virtuálních sekvencerů a mnoha dalších technik. Samotná architektura projektu je zobrazena na obrázku č. 3.1. Hlavní komponentou je *testbench*, ve kterém jsou následně definovány všechny ostatní komponenty i DUT. DUT neboli *design under test* je označení aktuálně verifikované komponenty. DUT je připojen ke komponentám pomocí rozhraní, skrze které jsou každý takt hodinového signálu čteny vstupy DUT, nasimulována HDL implementace a následně nastaveny výstupy DUT. K rozhraním DUT jsou připojeny komponenty simulující funkci zařízení, které s DUT nějakým způsobem komunikuje. Například do DUT zapisuje nová data. Všechny komponenty simulující vnější zařízení jsou obaleny v UVM prostředí. Hlavní UVM prostředí obsahuje většinou kromě těchto komponent i scoreboard, který implementuje funkcionality samotného DUT pomocí modelu. Také porovnává chování tohoto modelu a skutečné chování DUT. UVM Test, který obaluje hlavní UVM prostředí, obsahuje ještě sekvence, které se budou spouštět na UVM sekvenceru specifického agenta v prostředí. Sekvence jako takové právě napodobují chování okolních zařízení. V rámci UVM testu je také konfigurační soubor, pomocí kterého se konfigurují komponenty a celé UVM prostředí. UVM testbench může obsahovat více testů, ale v jednu chvíli je spuštěn pouze jeden. V následujících kapitolách popíši pouze komponenty a techniky potřebné pro implementaci prostředí zajímavých v rámci této práce.



Obrázek 3.1: Architektury UVM projektu

Pomocí modelování na úrovni transakcí dokážeme tvořit znovupoužitelné moduly, které se mohou starat každý o jinou funkcionalitu nějakého protokolu. Komunikace mezi jednotlivými komponentami tedy probíhá pomocí předávání transakcí. Transakce je objekt třídy, který obsahuje veškeré informace potřebné ke komunikaci mezi dvěma komponentami. Transakce obsahuje proměnné, podmínky proměnných a další funkce, které můžeme provádět nad transakcemi. Pro předávání transakcí se využívají UVM porty a UVM exporty. UVM porty zajišťují odesílání transakce z komponenty a UVM exporty zajišťují příjem transakce. Z těchto exportů můžeme následně transakce vyčítat. Dá se také využít vestavěných zásobníkových propojů. Tyto zásobníkové propoje dokáží uchovávat více zaslaných transakcí od portu, čímž vytváří vyrovnávací paměť. To je výhodné zvláště, když export odebírá transakce pomaleji, než je port odesílá.

Existují také analyzační propoje. Tyto propoje fungují stejně jako již dříve zmiňované propoje, s tím rozdílem, že na jeden analyzační port může být připojeno více analyzačních exportů. Když pak odešle analyzační port transakci na rozhraní, tak kopii této transakce dostanou všechny zaregistrované analyzační exporty.

Testbench

V UVM se testbench stará o vytvoření objektu test a modulu DUT. Také má na starost vytvoření rozhraní mezi testem a DUT, které se stará o správné propojení signálů. Test je v UVM dynamicky inicializován za běhu programu a dovoluje tak běh různých testů bez nutnosti překladu mezi běhy jednotlivých testů. Pokud ale chceme měnit parametry, jako například šířka přenášeného slova na rozhraní, tak je nutné verifikaci přeložil pokaždé, když se parametry změní.

Test

Test má tři hlavní funkce v rámci UVM. Těmi jsou inicializace hlavního UVM prostředí a následná konfigurace. Po inicializaci má za úkol aplikaci stimulů na DUT

pomocí spuštění sekvencí. Jednotlivé testy se většinou aplikují pomocí dynamické změny nastavení prostředí a změny spouštěných sekvencí.

Prostředí

Prostředí je zodpovědné za obalení ostatních komponent do jednoho celku. V prostředí může být inicializováno další prostředí, které následně může obsahovat mnoho dalších komponent.

Konfigurace objektů

Konfigurační objekt se většinou objevuje v rámci prostředí a zodpovídá za dynamické nastavení určitých prvků prostředí a objektů. Příkladem může být název prostředí. Konfigurační objekt může být konfigurován od konfiguračního objektu rodičovského prostředí. Dokážeme tak nastavit celou verifikační strukturu pomocí konfiguračního objektu z hlavního UVM prostředí.

Sekvenční položka

Sekvenční položka je objekt obsahující data a pomocné funkce. Sekvenční položka slouží jako transakce předávaná mezi jednotlivými UVM komponentami. Jedná se tak o objekt, pomocí kterého si mezi sebou komponenty předávají nutné informace. Data v sekvenčních položkách mohou být jak ručně nastavované, tak náhodně generované. Dají se jim také nastavit podmínky pro generování. Pomocí toho se dá určit podmnožina hodnot určité proměnné, kterou chceme generovat.

Driver

Driver přijímá transakce vygenerované v sekvenci pomocí jemu přiřazeného sekvenceru. Tyto transakce následně nasazuje na rozhraní DUT. Dokáže také číst data z rozhraní a vracet je sekvenci přes sekvencer pomocí odpovědí. Tím dokážeme reagovat na aktuální stav sběrnice a upravovat generování nových stimulů.

Monitor

Monitor vzorkuje data na sběrnici a převádí je do transakcí. Následně je distribuuje do zbytku prostředí většinou pomocí analyzačního exportu. Na tento export může být napojen scoreboard a sběrač pokrytí dat.

Sekvence

Sekvence zodpovídá za podmíněné generování stimulů pro DUT. Sekvence mohou generovat jednu až nekonečno transakcí. Většinou se spouští na začátku verifikace, ale pokud to situace vyžaduje, tak dokážeme zařídit dynamické spouštění různých sekvencí v průběhu testu. Sekvence musí být navázána na určitý sekvencer, aby se zajistil správný tok transakcí.

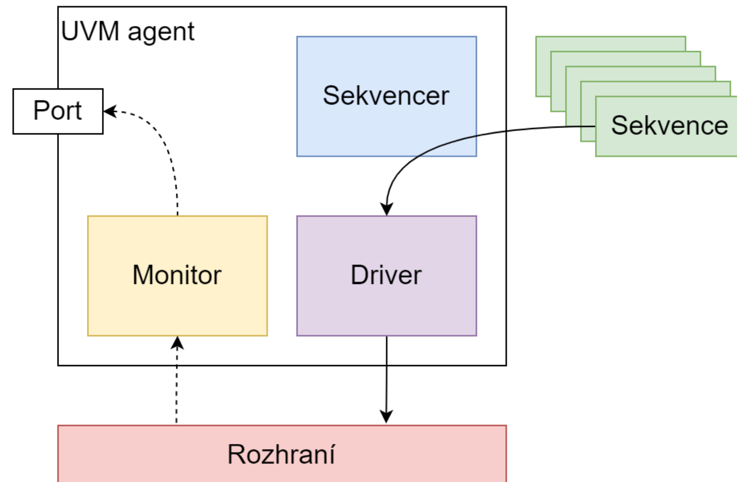
Sekvencer

Sekvencer se stará o správné předávání transakcí, které byly vygenerovány pomocí sekvencí. Většinou nemívá žádnou další funkcionalitu.

Agent

Agent je komponenta zodpovědná za komunikaci s určitým rozhraním, nebo s nějakým dalším agentem. Jedná se o hierarchickou komponentu, ve které jsou obsaženy další

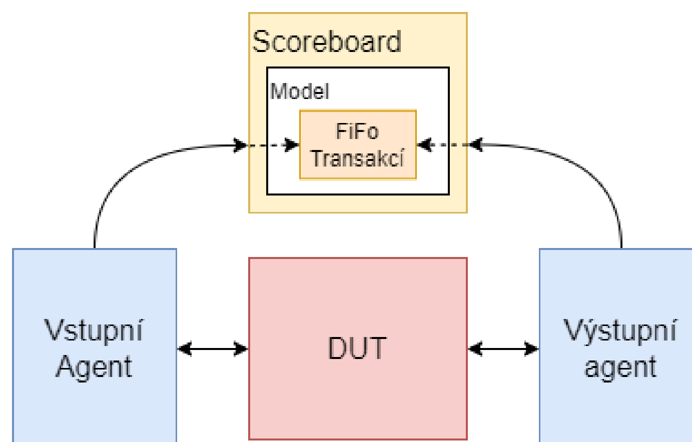
komponenty zodpovídající za určitou funkcionalitu v rámci komunikace s rozhraním. Tyto komponenty jsou sekvencer, driver, monitor a konfigurační objekt. Agent může fungovat ve dvou módech. Prvním módem je aktivní mód, který se používá, pokud je agent zodpovědný i za generování stimulů pro DUT. Druhým módem je mód pasivní, kdy agent pouze reaguje na aktuální stav sběrnice.



Obrázek 3.2: Struktura UVM agenta

Scoreboard

Scoreboard zodpovídá za kontrolu chování DUT. Snaží se predikovat, jaký bude výstup DUT při aplikaci vstupních vektorů. Získává transakce pomocí analyzačních portů agentů jak ze vstupních, tak z výstupních rozhraní. Propojení do scoreboardu je zobrazeno na obrázku 3.3, kde vlevo je vstupní agent a vpravo výstupní. Predikovaná data následně porovnává s těmi, co dostal z výstupních agentů. Verifikace zachytí chybu, pokud nejsou výstupní vektory stejné, jako ty predikované. Predikce probíhá na základě referenčního modelu.



Obrázek 3.3: Propojení scoreboardu

Vrstvení agenti

Vrstvenými agenty jsem nazval techniku rozdělování funkcionalit složitého rozhraní na pod agenty. Dosáhneme tak modulárních agentů, u kterých se každý stará jen o malou část funkcionality.

Virtuální sekvencer

Virtuální sekvencer zajišťuje spouštění různých sekvencí na náležitých sekvencerech. Slouží také na synchronizaci spouštění veškerých sekvencí. Můžeme pomocí něj nasimulovat různé scénáře chování z hlediska celého DUT a nejen v rámci jednotlivých rozhraní DUT.

Knihovna sekvencí

Knihovna sekvencí seskupuje veškeré sekvence určené pro určitou komponentu. A následně přes ni můžeme tyto sekvence spouštět v různém pořadí a s různými parametry za běhu programu. Dokážeme díky ní spouštět jak nějakou určitou sekvenci, kterou si zvolíme, tak dokáže zajistit i náhodné spouštění jedné či více sekvencí za sebou.

Kapitola 4

Multi-sběrnice

Kvůli stále větší rychlosti komunikace přes internetovou síť a na ostatních rozhraních v oblasti výpočetních technologií jsou vývojáři FPGA designů nuceni využívat čím dál tím širší paralelní sběrnice pro zvýšení rychlosti zpracování informací. Použití širokých sběrnic má ale jednu velkou nevýhodu, a to vysokou režii při zarovnávání přenášených slov. Zvláště při přenosu velice krátkých rámců. Tuto režii dokáží minimalizovat multi-sběrnice [8] pomocí zavedení kontrolních signálů určující začátky a konce jednotlivých přenášených rámců. Také zajišťuje rozdělení přenášeného slova do různých logických celků zajišťující snížení režie při zarovnávání.

Nejdůležitější výhodou multi-sběrnic je schopnost přenášet více efektivních hodnot v jednom hodinovém taktu. Dosahují toho pomocí přenášení jak datových linek, tak i linek pro předání kontextu dat uložených na datových linkách. V rámci této práce se budeme zabývat sběrnicemi *Multi Value Bus* (MVB) a *Multi Frame Bus* (MFB), které jsou blíže popsány v následujících podkapitolách.

Na obou sběrnicích se komunikuje jednosměrně. To znamená, že vysílající strana je zodpovědná za nastavení dat, která se mají přenést k příjemci. Příjemce je na druhou stranu zodpovědný za vyčtení dat ve správný moment. Tento moment nastává tehdy, až obě strany nastaví signály připravenosti do logické jedničky. U přijímací strany se tento signál označuje jako DST_RDY a u odesílající strany se označuje jako SRC_RDY. Po nastavení dat od odesílající strany data zůstávají na sběrnici, dokud není připravena pro přenos i strana přijímající.

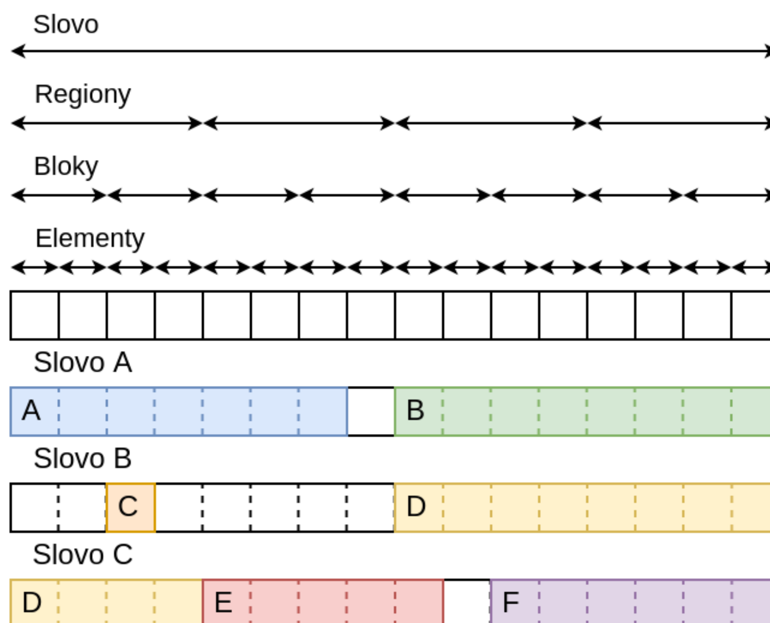
4.1 Specifikace MFB

Sběrnice Multi Frame Bus poskytuje způsob, jak přenášet mezi odesílatelem a příjemcem více různě dlouhých rámců. Přenášené slovo je rozděleno do regionů. V rámci jednoho regionu může začínat a končit vždy pouze jeden rámec. To znamená, že maximální počet přenášených rámců v jednom slově je roven počtu regionů. Dále jsou regiony rozděleny do bloků. Bloky slouží jako další zjednodušení při zarovnávání dat. A to kvůli tomu, že nový rámec může začínat pouze na začátku některého z bloků. Bloky jsou následně rozděleny do elementů. Konce rámců mohou být zarovnány s jakýmkoliv z elementů. Konec rámců omezuje pouze podmínka jednoho konce rámce na region. Na obrázku č. 4.1 můžete vidět ilustrační komunikaci v rámci tří přenesených slov. Kde jsou jednotlivé rámce různě dlouhé a označeny různými písmeny. Rámce jsou barevně rozlišeny pro jednodušší identifikaci. Z obrázku je patrné, že jednotlivé rámce mohou zasahovat i do více slov. Například rámec

D začíná v půlce slova B a končí až ve čtvrtině slova C. Sběrnice dokáže také přenášet metadata pro každý přenášený rámec. Sběrnice se dá nastavit tak, aby se metadata vůbec nepřenášela. U sběrnice se nachází pro každý region položka pro metadata. Kvůli možnosti konce i začátku rámce v jednom regionu je nutné mít zavedená jasná pravidla, kde se metadata pro každý z rámců přenáší. Může to být buď v metadatové položce regionu, kde rámec začíná anebo v metadatové položce, kde rámec končí. Tak nikdy nedojde ke kolizi metadat.

Sběrnice může mít různé nastavení. Sběrnice se dá nastavit pomocí následujících parametrů. Parametr R, který udává počet regionů. Parametr RS určující počet bloků v regionu. Parametr BS, který určuje počet položek v bloku. A parametr IW, který určuje šířku položky v bitech. Délka přenášeného slova je určena jako R*RS*BS*IW. Nastavení sběrnice se pak projeví i v šířce signálů ovládajících samotnou sběrnici. Na obrázku č. 4.3 je zobrazen přenos dat na MFB sběrnici, která má čtyři regiony, dva bloky na region, dva elementy na blok a nedefinovanou šířkou elementu.

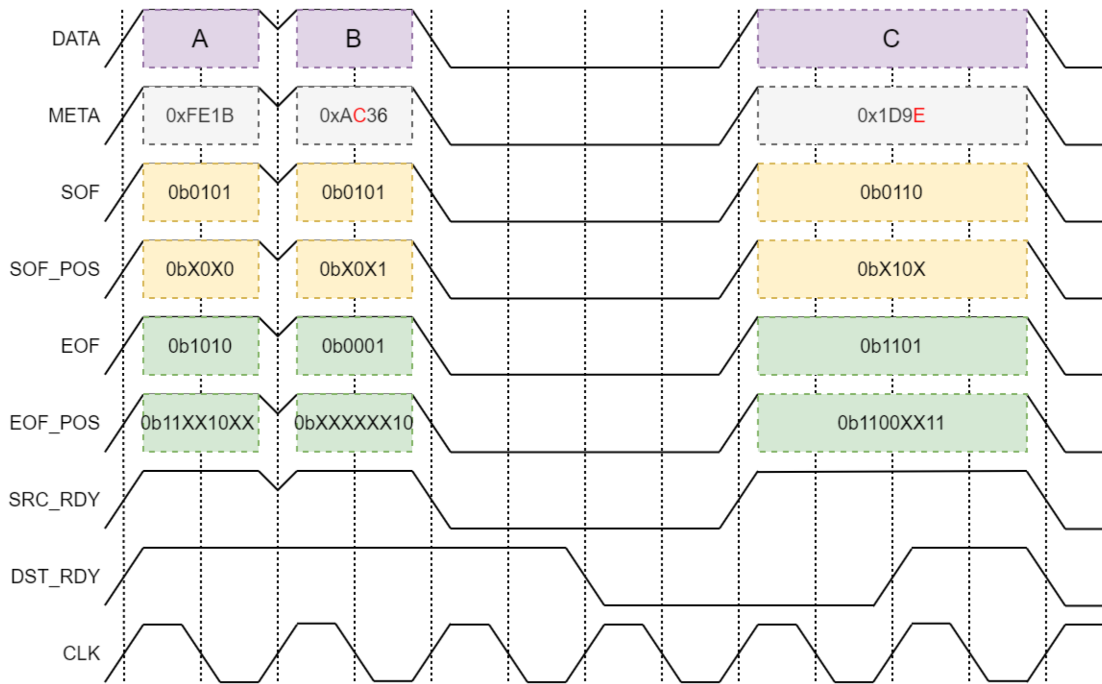
V rámci MFB se samotná data přenášejí pomocí vektoru DATA. Metadata pro data jsou přenášeny skrz vektor META. Potřebujeme také vědět, v kterých regionech rámce začínají a v kterých končí. K tomuto účelu slouží vektory SOF a EOF. Tyto vektory mají pro každý region jeden bit, určující jestli v regionu začíná/končí rámce. Musíme být také schopní zjistit, na kterém bloku v rámci regionu rámec začíná. K tomu slouží vektor SOF_POS. Stejně jako pro začátek rámce musíme dokázat zjistit, na které položce v rámci regionu rámec končí. K tomu slouží vektor EOF_POS.



Obrázek 4.1: Přenos dat přes MFB

Na obrázku č. 4.2 ilustruji přenos dat zobrazených na obrázku č. 4.1. Komunikace přenáší 3 slova obsahující dohromady 6 rámců. V průběhu signálů jsou nedefinované signály označeny jako "X". U přenosu prvního slova předpokládáme, že se jedná o zcela novou komunikaci. To znamená, že rámec A je prvním přenášeným rámcem. Z hodnoty 0b0101 nastavené na signálu SOF zjistíme, že v rámci prvního slova začínají dva rámce. První

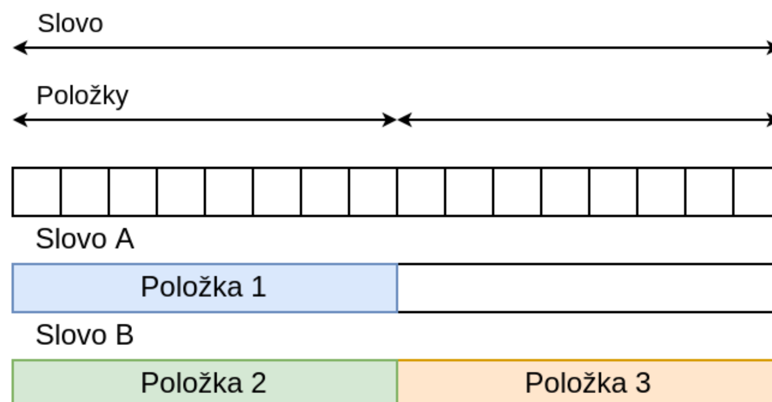
rámec v nultém regionu a druhý rámec v druhém regionu. Z hodnoty 0bX0X0 na signálu SOF_POS zjistíme, že první rámec začíná na nultém bloku v nultém regionu. A druhý rámec začíná na nultém bloku v druhém regionu. Stejným způsobem si zjistíme i konce rámců. Rozdílem je, že, pro výpočet pozice konce rámce použijeme vždy dva bity pro jeden region. První rámec končí na druhé položce prvního regionu a druhý rámec končí na třetí položce třetího regionu. Stejný postup použijeme pro výpočet začátků a konců všech ostatních rámců. Zajímavý je pro nás ještě přenos rámce D, který začíná ve slově C na pozici nultého bloku druhého regionu. A konec tohoto rámce se objevuje až ve slově C na pozici třetí položky nultého regionu. Z průběhu komunikace také dokážeme zjistit metadata přiřazená k jednotlivým rámcům. Pro každý region se v tomto případě přenáší jedno hexadecimální číslo. Pokud by se přenášela metadata na začátku rámce, tak by pro rámec D byla hodnota jeho metadat rovná 0xC přenesená v rámci metadat ve slově B. V opačném případě by měla metadata hodnotu 0xE přenesená v rámci metadat ve slově C.



Obrázek 4.2: Průběh MFB komunikace

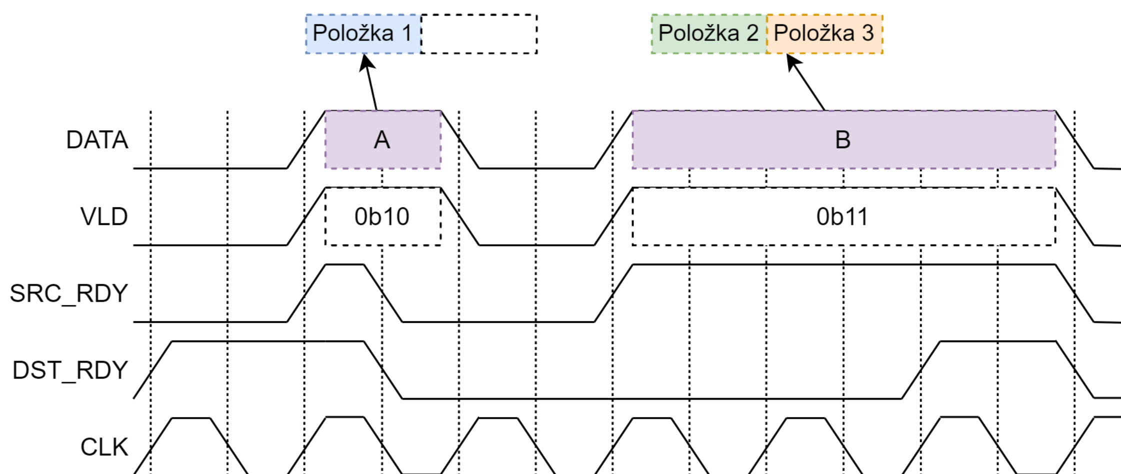
4.2 Specifikace MVB

Multi Value Bus je oproti MFB velice jednoduché rozhraní. Slouží pro přenos většího množství stejně dlouhých položek. Položky se za sebe skládají do slova. V jednom slově mohou být přeneseny jak validní, tak nevalidní položky. Validitu položek dokážeme zjistit díky pomocným signálům. Pro každý rámec je přiřazen jeden bit, který určuje platnost dané položky. Tyto validační bity jsou uloženy v rámci vektoru, který je označen jako VLD. Data jsou přenášena ve vektoru, který je označen jako DATA. Sběrnice MVB má také dva parametry. Parametr I určuje počet položek ve slově a parametr IW určuje šířu položky v bitech. Délka přenášeného slova je určena jako I*IW. Na obrázku 4.3 je zobrazen přenos dat s který má dvě položky na jedno slovo a délkou jedné položky rovné jednomu bajtu.



Obrázek 4.3: Přenos dat přes MVB

Na obrázku č. 4.4 ilustruji přenos dat zobrazených na obrázku č. 4.3. V rámci komunikace se přenáší dvě slova A a B. Na začátku komunikace se nastaví signál připravenosti přijímače do logické jedničky. Přenos prvního slova se provede pouze pokud jsou obě strany přenosu připraveny. V druhém cyklu hodinového signálu se nastaví jak datový vektor, tak i vektor obsahující validační bity. Hodnota validačního signálu je rovna 0b10. Tato hodnota znamená, že je validní pouze horní položka v přenášeném slově. Tedy položka 1, která je obarvena modře. Také se nastaví signál připravenosti od odesílající strany. Při přenosu druhého slova nastane zpracování hodnot od vysílací strany jako první. Tyto hodnoty se musí na rozhraní držet do té doby, než přijímací strana nastaví svůj signál připravenosti k přenosu. Hodnota validačního vektoru u druhého přenášeného slova je rovna 0b11, což značí validitu všech položek ve slově. Takže ve slově B jsou přeneseny položky 2 a 3, které jsou zvýrazněny oranžovou a zelenou barvou v tomto pořadí.



Obrázek 4.4: Průběh MVB komunikace

Kapitola 5

Tvorba verifikačních prostředí

Při návrhu prostředí jsem se zaměřil především na jednoduché použití komponent a rozšířitelnost. Následující kapitola popisuje návrh a funkcionalitu verifikačních prostředí. Také popisuje funkcionalitu jednotlivých komponent a význam jejich použití v prostředí. Jak MVB, tak MFB prostředí fungují ve dvou módech. Ve vysílajícím a přijímacím. Vysílající mód obstarává generování dat a následné odeslání těchto dat na rozhraní. Také obsluhuje generování signálů zařizujících řízení rozhraní. Oproti tomu přijímací mód zařizuje pouze generování signálů zařizujících řízení rozhraní. Proud dat do a z DUT je zobrazen v obrázku č. 5.1.

Obrázek 5.1: Zobrazení proudu dat do a z DUT

5.1 Prostředí logic_vector

Tato komponenta bude pomocné prostředí, které bude použito v rámci prostředí MFB pro generování metadat. Prostředí logic_vector zajišťuje generování vektorů dat s předepsanou délkou. Tato délka je definována pomocí parametrů a nejde měnit za běhu testu. Metadata jsou v rámci verifikací většinou nepodstatná a stačí ověřit, že z DUT vyšla stejná data, jako se do něj poslala. Proto byly navrženy pouze základní dvě sekvence, které dostatečně pokryjí většinu potřeb. První sekvence generuje náhodná data. Tato sekvence nikdy sama od sebe nikdy neukončí generování. Konec generování náhodných dat nastane teprve, až skončí test. Druhou sekvencí jsou také generována náhodná data. Ale na rozdíl od první sekvence ukončí generování po dosažení určitého počtu vygenerovaných dat. Sekvence má horní a spodní hranici počtu generovaných dat. Tyto hranice se dají přenastavit. Po randomizaci sekvence je určen počet sekvencí, který je mezi těmito hranicemi. Prostředí je uzpůsobeno rozšířitelnosti a stačí implementovat další sekvenci, která pokryje další potřeby použití. Samotné prostředí obsahuje pouze agenta. Agent se skládá z monitoru, driveru a sekvenceru.

5.2 Prostředí byte_array

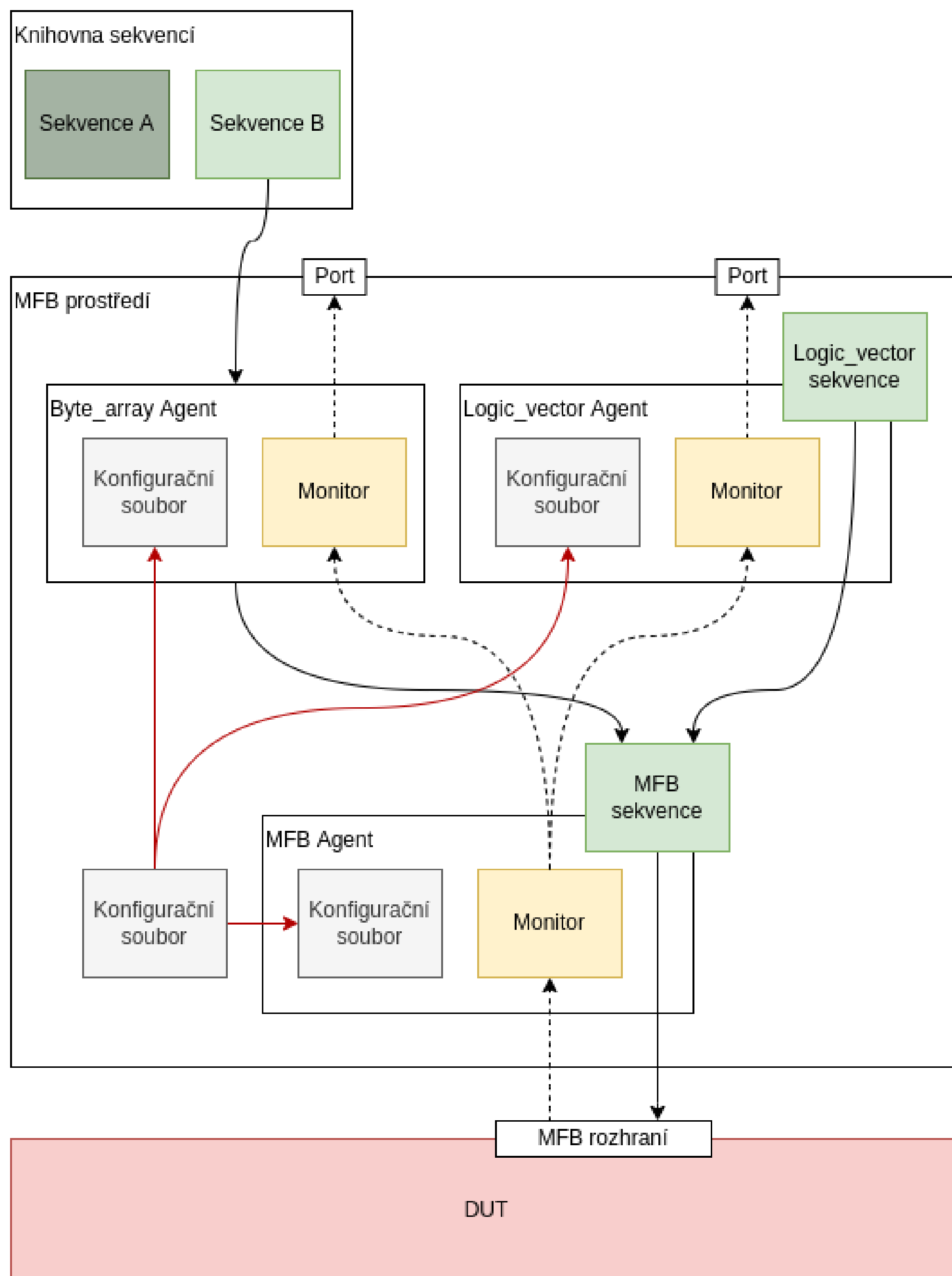
Tato komponenta je pomocné prostředí, které bude použito v rámci prostředí MFB pro generování datových rámců s různou délkou. Prostředí byte_array generuje náhodně velké pole osmi bitových vektorů. Pokud jednotlivé položky pole poskládáme za sebe, vytvoříme tak jeden dlouhý vektor s náhodnou délkou. To je velmi důležité, zvláště pro verifikační prostředí MFB sběrnice, kde budeme potřebovat ověřit přenášení různě dlouhých vektorů. U tohoto prostředí byly také implementovány dvě sekvence se stejnou funkcionalitou jako u prostředí logic_vector. Jediným rozdílem je, že se u těchto sekvencí nastavuje i dolní a horní hranice délky vektorů. To proto, abychom negenerovali zbytečně krátké, nebo dlouhé vektory. Kromě těchto sekvencí jsou implementovány i další specifičtější sekvence. Například sekvence generující stále stejně dlouhé vektory. Sekvence, které postupně zvětšují nebo zmenšují délky vektorů. A sekvence, u kterých hodnotu vektorů udává Gaussovo rozdělení. Prostředí má stejnou strukturu jako je u prostředí logic_vector. Pouze jsou zde jiné položky pro generování a jiné sekvence. Prostředí byte_array bylo již implementováno a já ho pouze využívám k dalšímu rozšíření UVM prostředí.

5.3 Multi Frame Bus (MFB)

U rozhraní MFB se mohou rámce rozdělit do více slov, které jsou přeneseny ve více než v jednom taktu. Proto budeme potřebovat vrstvené agenty. Agent MFB bude zajišťovat generování signálů pro MFB rozhraní v přijímacím režimu. Při nastavení rozhraní na vysílací mód se spustí sekvence, která odebírá a skládání data, od agentů z vyšší vrstvy. Zařizuje také správně nastavené řídících signálů a určování mezirámcových mezer. Propojení agentů je zobrazeno na obrázku č. 5.2. Struktura samotných agentů odpovídá popisu v kapitole 3.

Samotné generování přeposílaných rámců zajistí agent byte_array, který generuje pole o náhodné délce obsahující vektory dat se stejnou délkou. Prvky pole poskládané za sebe reprezentuje jeden celý rámec.

Rozhraní MFB může přenášet kromě dat i metadata. Tato metadata mohou být přenášena u začátku rámce, u konce rámce anebo nejsou přenášena vůbec. Chování generování je předem nastaveno a je obsaženo v konfiguračním souboru. Pomocí tohoto nastavení se mění funkcionalita MFB agenta. Generování metadat zajistí logic_vector agent. Tento agent generuje náhodný vektor dat s předem danou délkou. A pomocí konfigurace metadat se mění i chování monitorů a sekvencí.



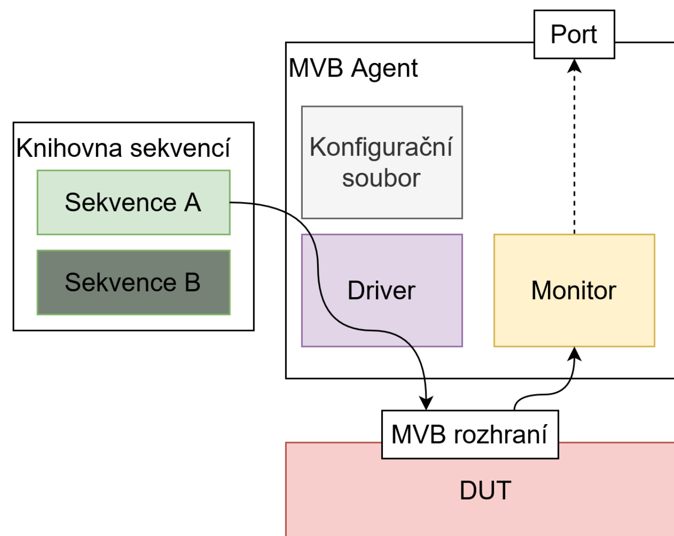
Obrázek 5.2: Diagram návrhu prostředí pro MFB rozhraní

Monitory logic_vector agenta a byte_array agenta jsou připojeny na port monitoru MFB agenta. Monitor MFB agenta sleduje aktuální stav na rozhraní. Tento stav uloží při každém kroku hodinového signálu do MFB sekvenční položky. Tyto položky následně vloží na analyzační port. Monitor logic_vector agenta odebere sekvenční položku z analyzačního portu a vyseparuje pouze metadata. Tyto metadata separuje podle nastavení prostředí. Následně tyto metadata odešle na svůj nastavený port. Monitor byte_array má stejnou funkcionalitu jako monitor logic_vector s tím rozdílem, že separuje místo přenesených metadat přenesená data. Každý agent si tak odebírá veškeré data a sám zodpovídá za správné

separování pro něj zajímavých dat. Model si následně může odebírat data a metadata nezávisle na sobě.

5.4 Multi Value Bus (MVB)

Rozhraní MVB sice přenáší více hodnot za jeden takt hodinového signálu, ale nenastává u něj přenos části dat v jednom slově a zbytek v dalších slovech. Proto v tomto rozhraní nebude potřeba vrstvených agentů. Bude zde stačit pouze jeden, který bude obstarávat jak generování čistých dat, tak i generování řídících signálů rozhraní. Diagram specifické struktury a propojení MVB agenta je zobrazen na obrázku č. 5.3 a jeho komponenty jsou popsány v následujících odstavcích. Kromě zobrazených komponent MVB agent obsahuje i komponenty popsané v teoretického popisu UVM agenta viz kapitola 3.



Obrázek 5.3: Diagram návrhu prostředí pro MVB rozhraní

Data jsou generována pomocí sekvencí uložených v knihovně sekvencí. Knihovna sekvencí obsahuje veškeré sekvence spouštěné v rámci testu. Sekvence se pouští na sekvenceru obsaženém v MVB agentovy. Pro vysílající mód je implementována jednoduchá sekvence generující náhodný počet MVB slov. Veškeré signály jsou randomizovány v rámci generování nové MVB sekvenční položky.

Dále prostředí obsahuje MVB agenta, který obaluje veškeré ostatní komponenty potřebné pro samotné fungování prostředí. První komponenta obsažená v MVB agentovi je konfigurační soubor. Konfigurační soubor obsahuje veškeré nastavení pro celého agenta. Je to například název agenta, způsob chování agenta a nastavení veškerých komponent.

MVB agent také obsahuje driver. Tento driver u MVB rozhraní nezastává pouze funkci nastavení dat ze sekvence. Stará se taky o sledování hodnot na rozhraní v každém hodinovém taktu, podobně jako monitor. Zaznamenaná data poté předá zpět sekvenci, která na základě těchto dat dokáže upravit generováních dat určených pro další hodinový takt.

U MVB rozhraní je úprava generovaných dat důležitá pro detekci sestupné hrany signálu DST_RDY. Při nastavené logické jedničce na pinu SRC_RDY u MVB agenta ve vysílajícím módu se nesmí na sběrnici změnit data do té doby, než bude signál DST_RDY opět nastaven do logické jedničky. Jednoduše řečeno, po vygenerování validních dat se obsah sběrnice

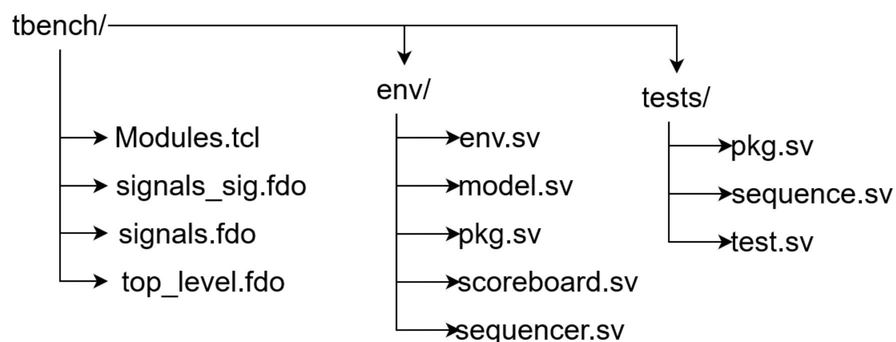
nemění do té doby, než je provedena transakce mezi odesílající a přijímací stranou. Tato funkcionalita vyplývá ze specifikace MVB rozhraní.

Poslední komponentou je monitor, který každý takt hodinového signálu získá aktuální hodnoty na všech pinech MVB rozhraní. Pokud se na obou pinech SRC_RDY a DST_RDY nachází logická jednička, tak odešle data na nastavený port agenta. Jelikož se v sekvencích generují vektory dat i validity jako celek, tak se na port zasílají vektory dat i validity. Tento port je většinou připojen na model ověřující správné fungování DUT. Správná interpretace validity dat je tedy přenechána na modelu.

Kapitola 6

Aplikace prostředí na tvorbu konkrétních verifikací

V této kapitole popisuji specifikaci, návrh a implementaci jednotlivých verifikací za použití verifikačních prostředí, která jsem navrhl a implementoval. Každá verifikace je rozdělena do více souborů, které se ještě následně dělí do různých složek. Struktura je zobrazena na obrázku č. 6.1. Hlavní složkou verifikace je složka tbench, ve které jsou obsaženy soubory pro překladový systém. Tyto soubory zařizují překlad a spuštění verifikace ve správném programu. Je zde také soubor Modules.tcl, který obsahuje cesty k veškerým komponentám, které budeme v rámci verifikace používat. Ve složce *tbench* jsou následně dvě podsložky. První složkou je *env*. Složka env obsahuje soubory, ve kterých je implementované verifikační prostředí a propojení jednotlivých komponent. V druhé podsložce *tests* jsou obsaženy všechny soubory obsahující nastavení samotného testu. V souborech pkg.sv jsou definovány parametry a balíčky souborů. Parametry uvedené v tabulkách u jednotlivých verifikací jsou nastavovány v rámci souboru tbench/tests/pkg.sv. Tyto parametry se následně předávají DUT při tvorbě prostředí. To se dělá kvůli tomu, aby bylo DUT nastaveno stejně jako verifikační prostředí.



Obrázek 6.1: Adresářová struktura verifikace

6.1 Komponenta asfifox

Komponenta asfifox[3] implementuje univerzální asynchronní fifo. To znamená, že zápis a čtení z komponenty probíhá časově nezávisle. Rozhraní pro zápis je časováno hodinami s jiným taktem, než rozhraní pro čtení. Pro komunikaci s komponentou se používá unikátní

sběrnice, na kterou se dá ovšem napojit s trochou snahy sběrnice MVB. A to jak na straně pro zápis, tak na straně pro čtení. Fifo dokáže ukládat pouze jednu položku. Do fifo fronty se tedy uloží vždy celé MVB slovo skládající se z jedné MVB položky. Komponenta je parametrizovaná. To znamená, že na základě nastavení parametrů mění svoji funkcionalitu. Parametry komponenty asfifox jsou zobrazeny v tabulce 6.1.

Jméno	**Datový typ**	**Výchozí hodnota**
ITEMS	Celé číslo	512
DATA_WIDTH	Celé číslo	512
RAM_TYPE	Řetězec	"BRAM"
FWFT_MODE	Pravdivostní hodnota	Pravda
OUTPUT_REG	Pravdivostní hodnota	Pravda
DEVICE	Řetězec	"ULTRASCALE"
ALMOST_FULL_OFFSET	Přirozené číslo	FIFO_ITEMS/2
ALMOST_EMPTY_OFFSET	Přirozené číslo	FIFO_ITEMS/2

Tabulka 6.1: Parametry komponenty asfifox

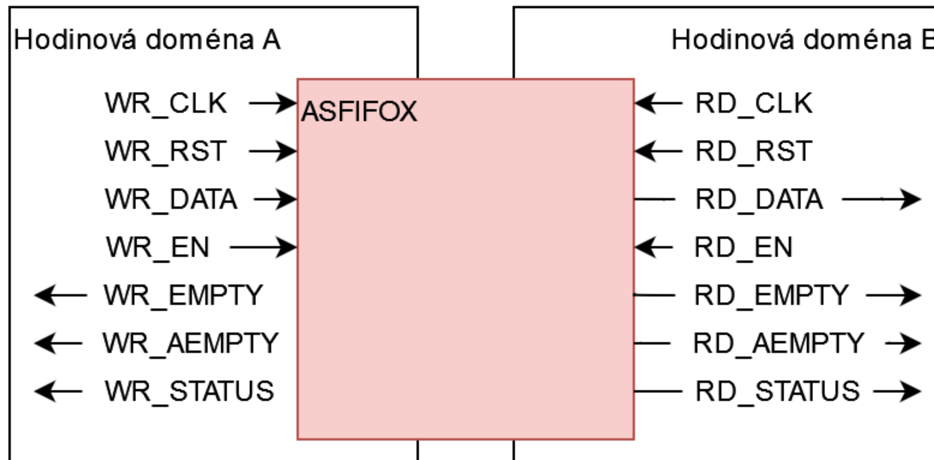
Parametr DATA_WIDTH udává šířku uložené položky. Šířka je udávána v bitech. Parametr ITEMS určuje maximální počet položek ve frontě, které musí být mocninou dvou. Minimální hodnota tohoto parametru je dva. Parametr RAM_TYPE udává typ paměti. Může být nastaveno na "BRAM", nebo "LUT". Parametr FWFT_MODE a OUTPUT_REG jsou přepínače zajišťující lepší časování v některých případech. Parametr DEVICE udává zařízení, na kterém bude komponenta aplikována. Komponenta asfifox tento přepínač aktuálně nevyužívá. A nakonec parametry ALMOST_FULL_OFFSET a ALMOST_EMPTY_OFFSET udávají odsazení počtu položek, kdy komponenta vyhodnotí, že je fronta skoro plná nebo skoro prázdná.

Komponenta má dvě rozhraní. První je určeno pro zápis a druhé pro čtení položek. Obě rozhraní mají v podstatě stejné signály. Výjimkou jsou signály, určující jestli je fronta skoro plná anebo skoro prázdná. V tabulce 6.2 jsou zobrazeny veškeré signály a jejich směry. Pokud je směr označen jako vstup, tak tento signál jde od zdroje signálu do komponenty asfifox. A pokud je směr označen jako výstup, tak je to přesně naopak. Pokud má signál prefix "WR_", je součástí rozhraní zodpovídající za zápis položek. A pokud má signál prefix "RD_", tak je součástí rozhraní zodpovídající za čtení položek. Výstupy WR i RD STATUS informují komponenty pracující s komponentou asfifox o aktuálním zaplnění vnitřní fronty. Výstup AFULL je jednobitový signál komunikující s komponentu, která položky do komponenty zapisuje. Informuje ji o tom, že se fronta zaplnila do takové míry, že je skoro plná. Výstup AEMPTY je také jednobitový signál. Tento signál ovšem komunikuje s komponentou, která vyčítá položky z komponenty asfifox a informuje o tom, že fronta bude již skoro prázdná.

Jméno	Směr	Šířka signálu[v bitech]
WR_CLK	Vstup	1
WR_RST	Vstup	1
WR_DATA	Vstup	DATA_WIDTH
WR_EN	Vstup	1
WR_FULL	Výstup	1
WR_AFULL	Výstup	1
WR_STATUS	Výstup	log2(ITEMS)
RD_CLK	Vstup	1
RD_RST	Vstup	1
RD_DATA	Výstup	DATA_WIDTH
RD_EN	Vstup	1
RD_EMPTY	Výstup	1
RD_AEMPTY	Výstup	1
RD_STATUS	Výstup	log2(ITEMS)

Tabulka 6.2: Signály komponenty asfifox

Na obrázku č. 6.2 je graficky zobrazeno rozložení jednotlivých hodinových domén. Hodiny WR_CLK jsou ovládány hodinovou doménou A. Hodiny RD_CLK jsou ovládány hodinovou doménou B. Všechny komponenty z jedné hodinové domény jsou ovládány stejným hodinovým signálem. Vstup WR_RST má informovat jednu polovinu komponenty asfifox o resetování komponenty do výchozího stavu. K tomu, aby se restart provedl korektně, musí být resetována i druhá strana komponenty pomocí signálu RD_RST. To znamená, že oba resetovací signály musí být nějakým způsobem synchronizovány, aby se resetování provedlo korektně. Pro korektní resetování komponent se dá navrhnou resetovací agent, který se dá zapojit do verifikačního prostředí. Tvorba tohoto rozhraní ovšem není součástí této práce.



Obrázek 6.2: Ilustrace komponenty asfifox

6.1.1 Verifikační plán

Po prostudování specifikace komponenty jsem přistoupil k tvorbě verifikačního plánu. Tento plán obsahuje veškeré funkcionality komponenty asfifox, které chci v rámci verifikace otes-

tovat. Verifikační plán je zobrazen v tabulce 6.3. Navržené cíle verifikace jsem vybral z následujících důvodů. Cíl na dostatečné pokrytí kódu jsem navrhl proto, abych se ujistil, že se provedlo velké množství příkazů FPGA komponenty. To je dobrý ukazatel správně vytvořené verifikace. Dostatečná variace vstupních dat ověří, že se komponenta chová korektně pro různé průchozí transakce. Je pak menší pravděpodobnost, že se komponenta zachová jiným způsobem, než bylo navrženo. A proto se v plánu objevuje cíl s názvem "Transakce". Ověření nekorektní změny ovládajících signálů je důležitý cíl. Nedodržení tohoto cíle by znamenalo, že se rozhraní komponenty nechová podle její specifikace. Z toho by vyplývalo, že je komponenta vadná. A cíle na zaplnění a vyprázdnění front jsou krajní případy užití. Při nesprávném ošetření těchto případů užití by se mohla komponenta například zaseknout. A kvůli tomu by se mohlo zastavit celé prostředí, kde by se komponenta použila.

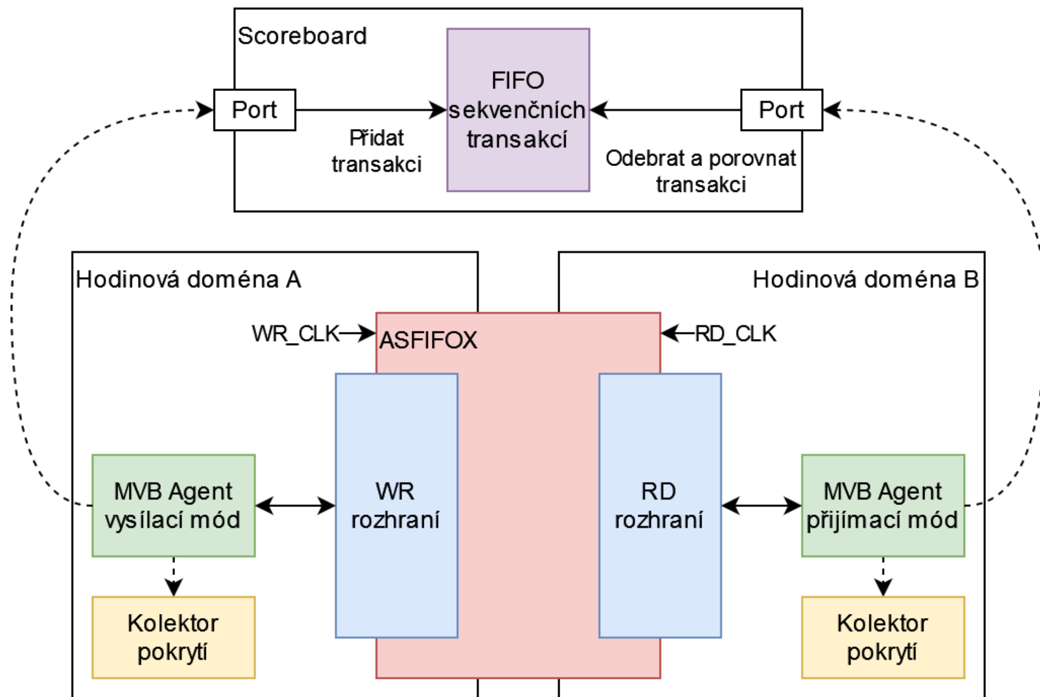
Sekce	**Název**	**Popis**	**Pokrytí**	**Priorita**
Kód komponenty	Kód	Alespoň 90 % kombinovaného pokrytí kódu	Kódu	1
Všechna rozhraní	Transakce	Dostatečná variace dat	Funkční	2
	RDY signály	Ověření nekorektní změn signálů připravenosti	Tvrzení	1
Vstupní rozhraní	Zaplnění	Ověření funkcionality při zaplnění vstupní fronty položkami	Funkční	1
Výstupní rozhraní	Vyprázdnění	Ověření funkcionality při úplném vyčtení položek z výstupní fronty	Funkční	1

Tabulka 6.3: Cíle verifikačního plánu pro verifikaci komponenty asfifox

6.1.2 Implementace

Na začátku implementace jsem navrhl verifikační prostředí podle již známých faktů. Toto prostředí je graficky znázorněno na obrázku č. 6.3. Z obrázku je vidět, že jsou na komponentu napojeny dva MVB agenti. Na zapisovací rozhraní komponenty je zapojen MVB agent, který bude nastaven na vysílající mód. Tento agent je zodpovědný za generování testovacích vektorů. Tyto vektory jsou následně přiřazeny na zapisovací rozhraní DUT a zároveň jsou odesílány na vstupní port ve scoreboardu. Druhý MVB agent, který bude nastaven na přijímací mód je zodpovědný za vyčítání dat z komponenty. Tedy je hlavně zodpovědný za generování signálů připravenosti. Data následně odesílá do výstupního portu ve scoreboardu. K oběma MVB agentům jsou připojeny kolektory transakcí, které zaznamenávají všechna data na vstupu i na výstupu. Data se rozřazují do košů podle hodnot. Na konci běhu verifikace se z těchto kolektorů vygeneruje zpráva o veškerých zapsaných a vyčtených transakcích. Scoreboard je velice jednoduchý a nevyžaduje implementaci modelu ani komparátoru dat. Scoreboard obsahuje frontu sekvenčních položek. Do fronty se přidají nová data při zápisu na vstupní port. Při vyčtení zapsaných dat z DUT na výstupní port se data porovnají s první položkou ve frontě. Pokud se sekvenční položky shodují, tak se inkrementuje čítač porovnaných transakcí. V opačném případě se vyvolá chyba, která ukončí verifikaci a vypíše chybovou hlášku a zobrazí porovnávané vstupní i výstupní transakce. Posledním důležitým úkolem bylo propojit specifické rozhraní komponenty s rozhraními MVB. U rozhraní určeného pro zápis položek do DUT jsem propojil negovaný signál WR_FULL se signálem DST_RDY MVB agenta. Také jsem propojil logický součin signálů SRC_RDY a VLD vysílacího MVB agenta na port WR_EN. Druhé rozhraní určené pro čtení z DUT

jsem propojil negovaný signál RD_EMPTY se signály VLD a SRC_RDY MVB agenta napojeného na toto rozhraní. Pomocí tohoto zapojení můžeme s komponentou komunikovat stejně, jako kdyby měla MVB rozhraní s jednou položkou v přenášeném slově.



Obrázek 6.3: Prostředí pro komponentu asfifox

Po implementaci navrženého verifikačního prostředí a odstranění chyb v kódu jsem provedl první spuštění verifikace. První běh verifikace byl spuštěn s nemodifikovanými jednoduchými sekvencemi na vstupním i výstupním MVB agentovi. Při tomto běhu se do komponenty zapsalo 10 položek. Z toho se jich vyčetlo také 10 a v komponentě nakonec zůstala prázdná. Již první běh dosáhl dostatečného pokrytí kódu. Při vygenerování zprávy o pokrytí kódu bylo rovno 98.95 %. Při kontrole průběhu signálu jsem objevil, že nenastalo úplné zaplnění fronty. Zato nastalo úplné vyprázdnění fronty, což jsem poznal podle nastavení výstupního portu RD_EMPTY do logické jedničky. Pokrytí transakcí bylo po prvním běhu pouhých 58,33 %. Pro příští běh verifikace jsem se rozhodl o změně parametru ITEMS na 16, aby nastal případ úplného zaplnění fronty. Také jsem zvedl počet generovaných transakcí u vstupního MVB agenta na 100 000. A u výstupního MVB agenta na 200 000 transakcí. Při druhém běhu verifikace se do komponenty zapsalo 33342 položek. Byly vyčteny veškeré položky, takže v komponentě nakonec nezůstala žádná. Při kontrole průběhu signálů jsem zjistil, že nastalo zaplnění komponenty. Poznal jsem to díky tomu, že výstupní port WR_FULL byl v průběhu verifikace několikrát nastaven na logickou jedničku. Pokrytí transakcí bylo po druhém běhu 77,77 %. Veškeré důležité kombinace dat byly pokryty a tak jsem rozhodl, že je pokrytí transakcí dostatečné. Nepokryté kombinace dat jsou zapříčiněny tím, že se nepřenáší žádné nevalidní položky. V žádném z běhů programů nenastala nekorektní kombinace signálů připravenosti. V této fázi byly pokryty všechny cíle verifikačního plánu. To znamená, že není nutné další testování komponenty.

6.2 Komponenta MFB rozdělovač

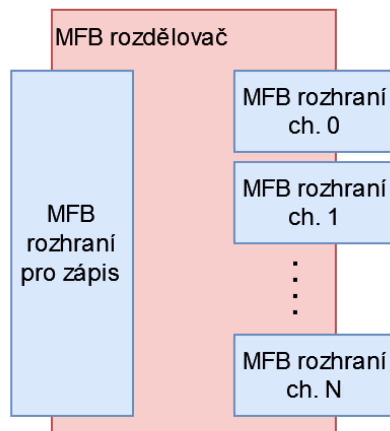
MFB rozdělovač[5] je komponenta, která dokáže rozdělit jeden vstupní proud MFB rámců na různá výstupní MFB rozhraní. Komponenta používá pouze MFB rozhraní pro komunikaci s okolním prostředím. Komponenta má parametry zobrazené v tabulce 6.4, které ovlivňují funkcionalitu. Parametr SPLITTER_OUTPUTS udává počet výstupních MFB rozhraní, na které se budou předávat vstupní rámce. Komponenta má také klasickou pětici parametrů pro nastavení MFB sběrnice. A parametr DEVICE udává zařízení, na kterém bude komponenta aplikována. V tabulce 6.5 jsou zobrazeny signály komponenty. Komponenta má tedy jedno vstupní MFB rozhraní. K tomto rozhraní je přidán selektovací vektor RX_MFB_SEL určující indexy výstupního rozhraní pro každý region. Určuje, na jaké výstupní MFB rozhraní má rámec putovat. Index pro jednotlivé regiony je validní, pokud je v regionu nastaven signál SOF do logické jedničky. Rámci může být přiřazena hodnota selektovacího vektoru, která musí být menší než log2(SPLITTER_OUTPUTS). Na výstupu komponenty je pole MFB rozhraní, na které putují vstupní rámce. Rozložení rozhraní komponenty je graficky zobrazeno na obrázku č. 6.4.

Jméno	**Datový typ**	**Výchozí hodnota**
SPLITTER_OUTPUTS	Celé číslo	8
REGIONS	Celé číslo	4
REGION_SIZE	Celé číslo	8
BLOCK_SIZE	Celé číslo	8
ITEM_WIDTH	Celé číslo	8
META_WIDTH	Celé číslo	1
DEVICE	Řetězec	"AGILEX"

Tabulka 6.4: Parametry komponenty MFB rozdělovač

Jméno	**Směr**	**Šířka signálu[v bitech]**
CLK	Vstup	1
RESET	Vstup	1
RX_MFB_SEL	Vstup	REGIONS*max(1, log2(SPLITTER_OUTPUTS))
RX_MFB	-	MFB rozhraní
TX_MFB	-	MFB rozhraní[SPLITTER_OUTPUTS]

Tabulka 6.5: Signály komponenty MFB rozdělovač



Obrázek 6.4: Ilustrace MFB rozdělovače

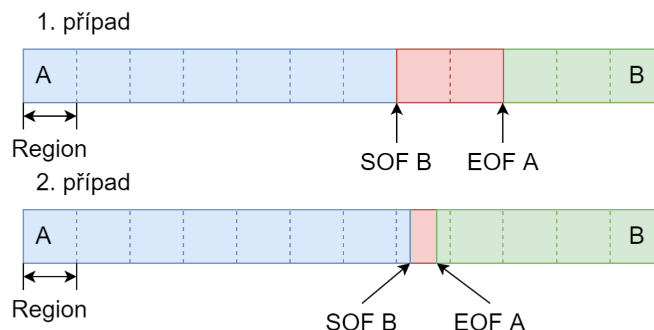
6.2.1 Verifikační plán

Při tvorbě verifikace jsem postupoval stejně jako u tvorby verifikace pro komponentu asfifox. Nejdříve jsem si vytvořil verifikační plán obsahující všechny funkcionality komponenty, které chci verifikací ověřit. Cíle verifikačního plánu jsou zobrazeny v tabulce 6.6. První dva cíle jsou stejné jako u verifikačního plánu pro komponentu asfifox. Kontrola signálů připravenosti by měla fungovat u MFB prostředí stejně jako u MVB agenta.

Cíl s názvem "Zarovnání" je navržen z důvodu ověření správného zarovnání signálů SOF, EOF a také signálů SOF_POS a EOF_POS. Musí být ověřeno, že žádný rámec nezačíná před koncem předchozího rámce. To by mělo za následek kolizi rámců. Nastává hned několik případů, jak by k tomu mohlo dojít. Na obrázku č. 6.5 jsou zobrazeny dva ilustrační případy kolize rámců. Jednotlivá políčka na obrázku vyznačují jeden region. První případ zachycuje sekvenci dvou začátků rámců s ukončením prvního rámce až v regionu, který je součástí nového rámce. To znamená, že druhý začíná před ukončením původního rámce. Tento případ může nastat i při přenosu rámců ve více slovech. Druhý případ zachycuje začátek nového rámce i konec prvního rámce ve stejném regionu. V tomto případě je nutné ověřit, že původní rámec končí na položce, která ještě není součástí nového rámce.

Sekce	Název	Popis	Pokrytí	Priorita
Kód komponenty	Kód	Alespoň 90 % kombinovaného pokrytí kódu	Kódu	1
Všechna rozhraní	RDY signály	Ověření nekorektní změny signálů připravenosti	Tvrzení	1
	Zarovnání	Ověření kolize rámců	Tvrzení	1
	Transakce	Průchod dostatečného množství transakcí	Funkční	2
Vstupní rozhraní	Zaplnění	Ověření funkcionality při zastavení vnitřní zřetězené linky	Funkční	3
Výstupní rozhraní	Vyprázdnění	Ověření funkcionality při úplném vyčtení položek ze všech kanálů	Funkční	3

Tabulka 6.6: Cíle verifikačního plánu pro verifikaci komponenty MFB rozdělovače



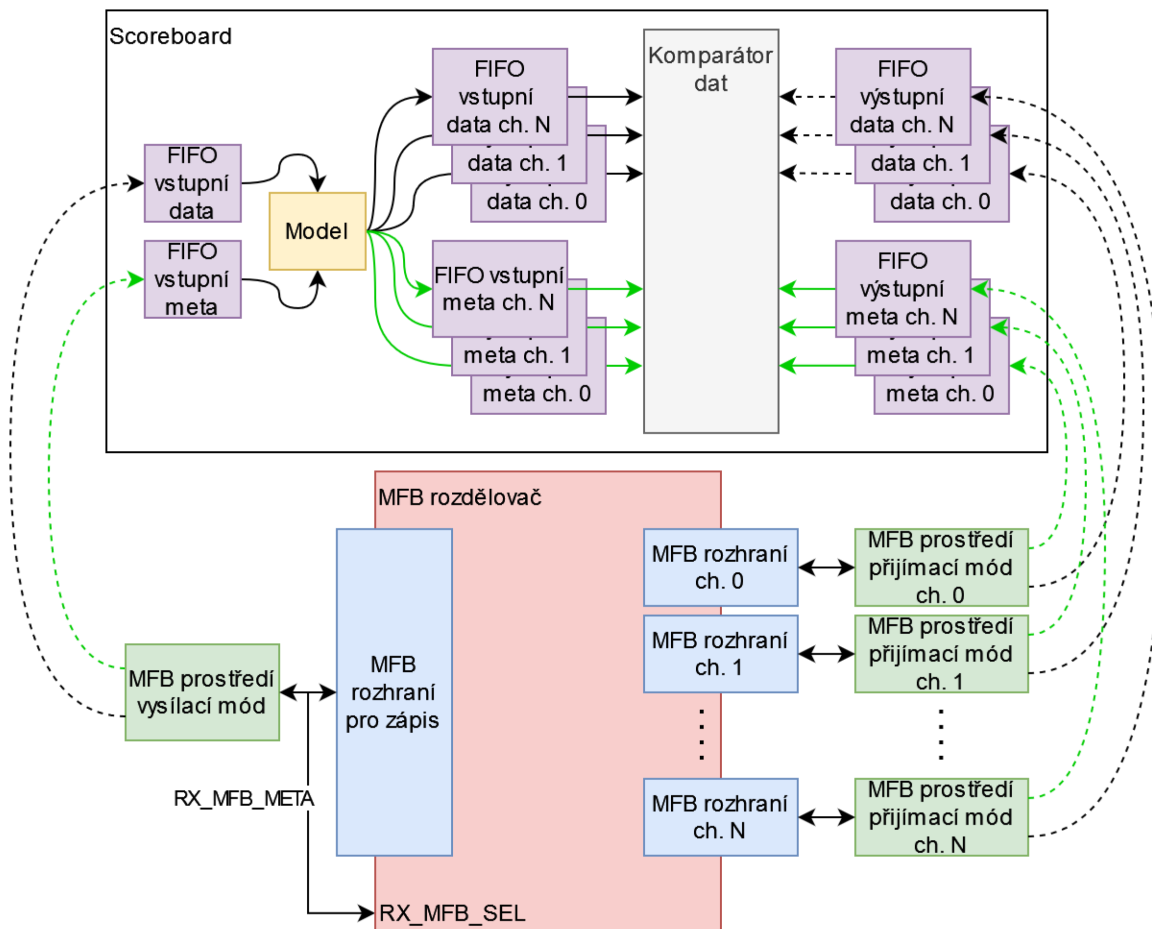
Obrázek 6.5: Kolize MFB rámců

Cíl s názvem "Zaplnění" ověřuje správnou funkcionalitu při pomalém či žádném odebírání rámců z výstupních rozhraní, což má za důsledek zastavení vnitřní zřetězené linky FPGA komponenty. Vyprázdnění všech kanálů naopak odebírá rámce rychleji, než jsou do komponenty posílány. Na tento případ použití je navržen cíl s názvem "Vyprázdnění". Obě tyto varianty funkcionalit mohou při nesprávné implementaci sabotovat práci prostředí, ve kterém bude komponenta umístěna. Proto je důležité tyto případy užití vyzkoušet. Cíl s názvem "Transakce"je navržen proto, aby komponentou prošlo dostatečné množství transakcí s různými hodnotami. Pokud komponentou projde dostatek transakcí bez chyby, tak by neměla selhat ani v reálném zapojení.

6.2.2 Implementace

Vstupní MFB prostředí je pouze jedno a k parametru pro šířku metadat je přidán dostatečný počet bitů pro rozpoznání indexu výstupního MFB rozhraní, udávající na které rozhraní má daný rámec putovat. To znamená, že z vygenerovaných metadat bude odseparována část, která bude předána na vstupní vektor RX_MFB_SEL. Toto řešení je nejjednodušší v případě, že nechceme tvořit speciální verifikační prostředí, které by rozšiřovalo MFB prostředí o generování selektovacího vektoru. Výstupních MFB prostředí je stejný počet jaký je daný parametrem SPLITTER_OUTPUTS. Všechna tyto prostředí jsou nastavena na přijímací mód a jsou propojena s korespondujícím výstupním MFB rozhraním. Všechna výstupní MFB prostředí mají spuštěnou nekonečnou smyčku náhodných sekvencí z knihovny sekvencí. Vstupní MFB prostředí také spouští sekvence ze své knihovny sekvencí. Počet těchto sekvencí se dá ovšem nastavit. Také je implementována speciální sekvence metadat, která generuje vstupní metadata s přidanými identifikátory pro určení indexu výstupních kanálů. Ve scoreboardu jsou pro každé MFB prostředí jedna fifo fronta určená na data a jedna určená na metadata. V rámci scoreboardu je vytvořen model simulující funkcionalitu MFB rozdělovače. V tomto modelu se ověří, jestli je index přiřazený k metadatům korektní. Pokud ano, tak vytvoří novou sekvenční položku metadat, která již neobsahuje index výstupního rozhraní. A následně přepošle data i metadata do výstupních front modelu patřící určitému výstupnímu rozhraní podle indexu. Pokud je ale index mimo hranice, tak se vypíše chybová hláška. Výstupem modelu jsou tedy dvě pole sekvenčních položek. Jedno s daty a jedno s metadaty. Pro výstupní MFB prostředí jsou ve scoreboardu stejně velká pole sekvenčních položek pro data a metadata. Ve scoreboardu jsou také komparátory dat pro každé výstupní prostředí zvlášť. Zásobníky z modelu i zásobníky určené pro sekvenční položky výstupních MFB prostředí jsou následně propojeny do korektního komparátoru podle indexu výstupního kanálu. V komparátoru se pak porovnávají data z modelu s daty od DUT a metadata

z modelu s daty od DUT. Pokud se data nebo metadata neshodují, tak verifikace končí a vypíše se chybová hláška. Komparátory dat také obsahují počet porovnaných rámců. To se hodí při výpisu po ukončení běhu verifikace. Zjednodušené prostředí je graficky zobrazeno na obrázku č.6.6. Jedná se o ilustrační obrázek pomáhající pochopit propojení komponenty DUT s verifikačním prostředím. Také je zde lehce naznačena funkcionalita scoreboardu.



Obrázek 6.6: Verifikační prostředí pro komponentu MFB rozdělovač

Po implementaci celého verifikačního prostředí přišla fáze odstraňování chyb a prvního spuštění. Při prvním spuštění jsem na vstupním MFB prostředí spustil náhodné sekvence z knihovnu sekvencí. Knihovně sekvencí jsem nastavil minimální počet spuštěných sekvencí na 50 a maximální počet spuštěných sekvencí na 70. Komponentou prošlo 5221 transakcí a jejich rozdělení podle výstupních kanálů je zobrazeno v tabulce 6.7. Po vygenerování zprávy o pokrytí kódu se ukázalo, že se mi podařilo hned při prvním běhu verifikace dosáhnout 100 % pokrytí. Při zkoumání průběhu signálu jsem přišel na to, že nastalo jak zaplnění vstupního MFB rozhraní, tak i vyprázdnění výstupních MFB rozhraní.

Kanál	Počet transakcí
0	662
1	680
2	629
3	649
4	673
5	669
6	640
7	619

Tabulka 6.7: Počet průchozích transakce rozdělené do výstupních kanálů při prvním běhu

Pro dostatečné otestování komponenty jsem verifikaci spustil ještě jednou se změněným počtem spouštěných sekvencí. Spodní hranici jsem nastavil na 200 sekvencí a horní hranici na 250 sekvencí. Také jsem snížil počet výstupních MFB rozhraní pouze na 2, pro lepší orientaci. A změnil jsem nastavení zarovnávání metadat na EOF. Verifikace proběhla úspěšně, což znamenalo úspěšné splnění všech cílů verifikace. Komponentou v tomto běhu proběhlo 19027 transakcí. Rozdělení transakcí do kanálů je zobrazeno v tabulce 6.8. Verifikace by měla aktuálně dostatečně ověřovat správnost komponenty a měla by být připravena na dodatečné rozšíření v případě rozšíření funkcionality samotné komponenty.

Kanál	Počet transakcí
0	9589
1	9438

Tabulka 6.8: Počet průchozích transakce rozdělené do výstupních kanálů při druhém běhu

6.3 Komponenta metadata extraktor

Komponenta metadata extraktor[4] přijímá MFB rámce a extrahuje metadata k nim přiřazeným. Metadata jsou následně přeposílána na výstupní MVB rozhraní. Samotná vstupní MFB slova jsou přeposílána beze změn na výstupní rozhraní MFB. Komponenta tedy má dvě MFB a jedno MVB rozhraní. Graficky je komponenta zobrazena na obrázku č. 6.7. Komponenta se dá nastavit pomocí parametrů vypsaných v tabulce 6.9. V této komponentě se dají nastavit jednotlivá specifika MFB a MVB rozhraní stejně jako u předchozích komponent. Jediným rozdílem je to, že parametry mají v názvu prefix podle toho, ke kterému rozhraní patří. Pro verifikaci pravděpodobně nejdůležitější nastavení je parametr EXTRACT_MODE, který udává, jestli se metadata přenášení na začátku rámce anebo na konci rámce. Má také parametry, určující na jakém zařízení bude komponenta nasazena. Také se dají zapnout roury pro MFB a MVB rozhraní. V tabulce 6.10 jsou zobrazeny veškeré signály s vypočítanou šířkou signálů.

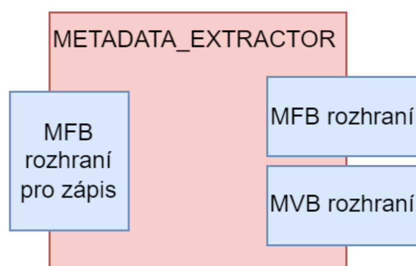
Jméno	**Datový typ**	**Výchozí hodnota**
MVB_ITEMS	Celé číslo	2
MFB_REGIONS	Celé číslo	2
MFB_REGION_SIZE	Celé číslo	1
MFB_BLOCK_SIZE	Celé číslo	8
MFB_ITEM_WIDTH	Celé číslo	32
MFB_META_WIDTH	Celé číslo	0
EXTRACT_MODE	Celé číslo	0
OUT_MVB_PIPE_EN	Logická hodnota	Nepravda
OUT_MFB_PIPE_EN	Logická hodnota	Nepravda
DEVICE	Řetězec	"ULTRASCALE"

Tabulka 6.9: Parametry komponenty metadata extraktor

Jméno	Směr	Šířka signálu[v bitech]
CLK	Vstup	1
RESET	Vstup	1
RX_MFB	-	MFB rozhraní
TX_MVB	-	MVB rozhraní
TX_MFB	-	MFB rozhraní

Tabulka 6.10: Signály komponenty metadata extraktor

Z tabulky signálů je zřejmé, že vstupní MFB rozhraní je stejné jako výstupní MFB rozhraní. Šířka signálu TX_MVB_DATA je závislá na šířce vstupních metadat MFB rozhraní. Jedna metadatová položka vstupního MFB rozhraní má stejnou šířku jako datová položka ve výstupním MVB rozhraní. Komponenta jako taková je vcelku jednoduchá na pochopení a zapojení. Tuto komponentu jsem vybral jako třetí verifikaci hlavně proto, že obsahuje oba typy sběrnice. To znamená, že mohu vyzkoušet propojení obou sběrnic v jedné verifikaci.



Obrázek 6.7: Ilustrace komponenty metadata extraktor

6.3.1 Verifikační plán

Stejně jako u předchozích dvou verifikací jsem začal tvorbou verifikačního plánu. Cíle verifikačního plánu jsou zobrazeny v tabulce 6.11

Sekce	Název	Popis	Pokrytí	Priorita
Kód komponenty	Kód	Alespoň 90 % kombinovaného pokrytí kódu	Kódu	1
Všechna rozhraní	RDY signály	Ověření nekorektní změny signálů připravenosti	Tvrzení	1
	Zarovnání	Ověření správného zarovnání MFB rámců	Tvrzení	1
	Transakce	Průchod dostatečného množství transakcí	Funkční	1
Výstupní rozhraní	Vyprázdnění dat	Ověření funkcionality při úplném vyčtení položek z výstupního MFB rozhraní	Funkční	3
	Vyprázdnění metadat	Ověření funkcionality při úplném vyčtení položek z výstupního MVB rozhraní	Funkční	2
Vstupní rozhraní	Zaplnění	Ověření zotavení při zaplnění komponenty	Funkční	2

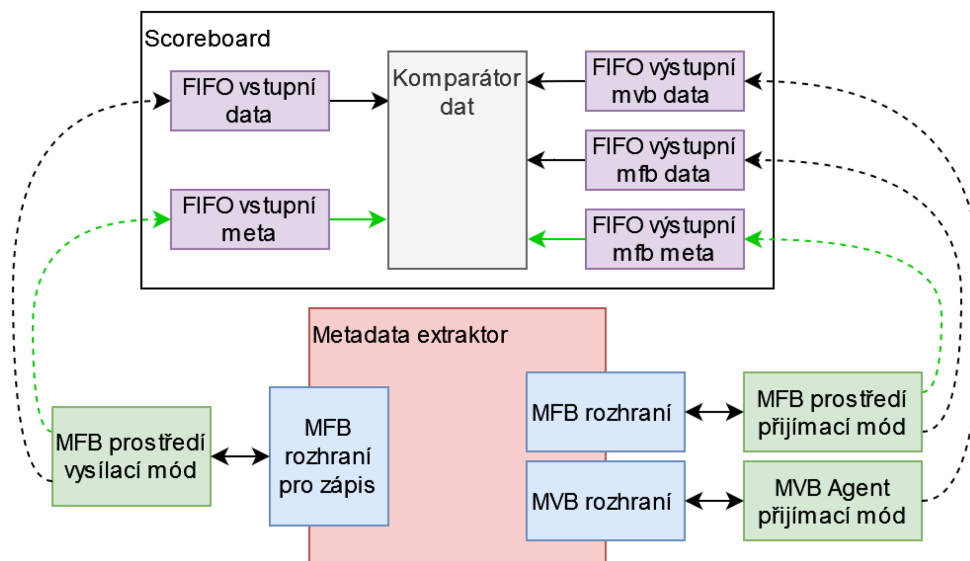
Tabulka 6.11: Cíle verifikačního plánu pro verifikaci komponenty metadata extraktor

V podstatě veškeré cíle jsou podobné a dosahují ověření stejných funkcionalit, jako v předešlých verifikacích. Nachází se zde cíl na pokrytí kódu. Ověření signálů připravenosti jak u MVB, tak u MFB sběrnic. Kontrola zarovnání rámců v MFB komunikaci. A cíle zaměřující se na zaplnění a vyprázdnění komponenty. Je vidět tendence toho, že se cíle vytvořené pro jednu verifikaci dají aplikovat i na další.

6.3.2 Implementace

Verifikační prostředí obsahuje dvě MFB prostředí a jednoho MVB agenta. První MFB prostředí bude nastaveno jako vysílající. A bude generovat metadata, která budou následně vyseparována a přeposlána na výstupní MVB rozhraní jako položky. Toto prostředí je také zodpovědné za generování vstupních rámců. Nastavení generování metadat v rámci vstupního MFB rozhraní musí odpovídat nastavení zarovnání metadat u samotné komponenty. Výstupním MVB rozhraní je zapojeno do MVB agenta nastaveného do přijímacího módu. Výstupní MFB rozhraní bude propojeno s MFB prostředím, které také bude nastaveno na přijímací mód.

Ve scoreboardu bude pět fifo front propojených s MFB prostředí a MVB agentem pomocí analyzačních portů. Do prvních dvou fifo front bude ukládat data a metadata vstupní MFB prostředí. Do dalších dvou fifo front bude ukládat data a metadata výstupní MFB prostředí. Poslední fifo fronta přijímá data od výstupního MVB rozhraní. Tato data reprezentují vyseparovaná metadata z MFB rozhraní. Díky jednoduché funkcionalitě testované komponenty nebudeme potřebovat model simulující funkcionalitu komponenty. Ve scoreboardu bude pouze komparátor dat, který vybere data a metadata z vstupních fifo front. Data i metadata porovná s těmi, která jsou ve výstupních fifo frontách pro MFB data a metadata. Pokud budou vstupní i výstupní MFB transakce stejné, tak porovná data z výstupního MVB prostředí a metadata ze vstupní fifo fronty pro metadata. Ilustraci navrženého prostředí je zobrazeno na obrázku č. 6.8.



Obrázek 6.8: Verifikační prostředí pro komponentu metadata extraktor

Po odstranění chyb v kódu jsem verifikaci poprvé spustil. Na výstupním MVB agentovi jsem nastavil spuštění jednoduché sekvence pořád dokola. Na výstupním MFB prostředí jsem spustil nekonečnou smyčku, kde se pouští vždy náhodná sekvence z knihovny sekvencí. Na vstupním MFB prostředí jsem nastavil hraniční počet spouštěných sekvencí. Horní hranici jsem nastavil na 30 sekvencí a spodní na 20 sekvencí. Verifikace proběhla v pořádku. Komponentou prošlo 2 095 transakcí. Po vygenerování zprávy pokrytí kódu jsem zjistil, že je míra pokrytí kódu na 72,67 %. A při průzkumu průběhu signálů jsem objevil, že nastalo jak vyprázdnění všech dat, tak i metadat z výstupních rozhraní. Bohužel nenastalo zaplnění vstupního rozhraní komponenty.

Pro druhý běh verifikace jsem zvýšil hranice spouštěných sekvencí. Horní hranici jsem nastavil na 200 a spodní na 150. Verifikace proběhla úspěšně a komponentou nyní prošlo 15 549 transakcí. Ze zprávy o pokrytí kódu jsem zjistil, že je míra pokrytí kódu na 78,12 %. Z důvodu malého nárůstu procent pokrytí kódu, jsem se rozhodl blíže prozkoumat co by to mohlo způsobovat. Zjistil jsem, že malý nárůst v procentech pokrytí kazí rekurzivní hierarchické pokrytí kódu FPGA komponent využívaných v rámci implementace komponenty metadata extraktor. Nás ale zajímá nejvíce pokrytí kódu lokální komponenty, které je 95,83 %, což je více než dostačující pro splnění našeho verifikačního cíle. Také nastalo zaplnění vstupního rozhraní komponenty. V rámci obou běhů verifikace neztroskotala na porušení tvrzení, takže můžeme předpokládat, že veškeré cíle verifikačního plánu pokrývající tvrzení jsou úspěšně splněny. Jelikož jsou veškeré cíle verifikačního plánu splněny, tak můžeme prohlásit verifikaci za dostačující.

Kapitola 7

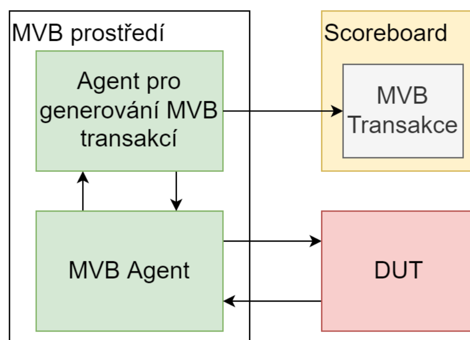
Vyhodnocení výsledků

Verifikační prostředí jsou funkční a použitelná v reálném nasazení. Podařilo se mi pomocí nich vytvořit tři funkční verifikace, které by se měly dát snadno rozšířit. Dají se také použít jako odrazový můstek při tvorbě nových verifikačních prostředí pro jiné sběrnice. Jako jeden z nedostatku bych uvedl propagaci veškerých signálů do scoreboardu od MVB agenta. Nedostatek je daný tím, že byla sběrnice navrhována jako první. Některé postupy jsem objevil až při návrhu a implementaci prostředí pro sběrnici MFB.

Pro všechny tři komponenty byly implementovány funkcionální verifikace. U všech verifikací byla ověřena jejich funkcionalita. Také u nich bylo dosaženo dostatečné pokrytí jak kódu, tak i funkcionální pokrytí. U žádné z nich nebyla objevena závada, což značí, že komponenty byly implementovány správně. Díky tomuto ověření můžeme komponentám důvěřovat natolik, aby byly nasazeny do reálného prostředí. Při začátku tvorby druhé a třetí komponenty se také ukázalo, že jsou části již vytvořené verifikace natolik obecné, že se dá převzít kód v podstatě jakékoliv verifikace a pomocí ní vytvořit verifikaci novou. Samozřejmě se při použití jiných sběrnic a jiných nastavení musí verifikační prostředí jinak propojit a nastavit.

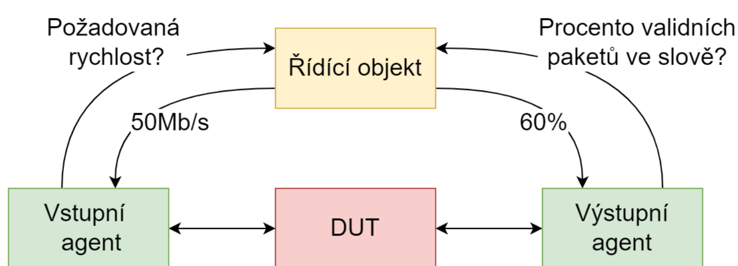
7.1 Možné rozšíření systému

Následující rozšíření by mohla pomoci při tvorbě nových verifikací zaměřených na pokrytí funkcionalit. Jako první rozšíření systému bych uvedl úpravu verifikačního prostředí pro MVB sběrnici. Pokud by se prostředí upravilo z jednoduchého agenta na prostředí s vrstvenými agenty, tak bychom mohli do komponenty scoreboard propagovat pouze platná data. To by do budoucna mohlo ulehčit implementaci modelů. Jelikož by se pokaždé ve scoreboardu nemuselo pořád dokola implementovat to stejné rozpoznávání platných a neplatných dat. Struktura nového prostředí pro MVB je zobrazena na obrázku č. 7.1.



Obrázek 7.1: Nové prostředí MVB rozhraní

Druhým rozšířením by mohl být konfigurační objekt. S tímto objektem by komunikovaly veškeré verifikační komponenty stejným způsobem pomocí předem nadefinovaných funkcí. Tento objekt by mohl pro nějakou část (například vstupní rozhraní komponenty) udávat například maximální propustnost rozhraní, kterou by předal zpět do prostředí v jednotné formě. Prostředí by pak pomocí této navrácené hodnoty dokázalo například upravovat generování vektorů. To by znamenalo, že tento objekt řídí rychlosti rozhraní, aby se daly lépe zacílit krajní případy užití, které jsou kritické na rychlost linky. Komunikace s navrhovaným konfiguračním objektem je zobrazena na obrázku č. 7.2.



Obrázek 7.2: Komunikace s konfiguračním objektem

Kapitola 8

Závěr

V rámci mé bakalářské práce se mi podařilo nastudovat a zdůvodnit použití technologie UVM. Práce zároveň ověřila přínos adoptování této technologie. Přínosem jsou zejména jasná struktura projektu, přehlednost, jednoduchost a jednoduchá znovupoužitelnost. Také se mi podařilo seznámit se s problematikou multi-sběrnic a funkcionalitou dvou specifických multi-sběrnic. Po nastudování specifikace těchto sběrnic se mi podařilo navrhnout a implementovat verifikační prostředí pro tyto sběrnice. Vytvořená prostředí jsou funkční a zároveň jednoduchá na další použití. Tyto poznatky jsem si ověřil na konci práce při implementaci verifikací za použití těchto prostředí. Samotná implementace prostředí může také sloužit jako základ při tvorbě nových prostředí pro další sběrnice.

Podařilo se mi také seznámit se s přístupem tvorby verifikací zaměřeným na pokrytí funkcionalit. Tento přístup spojený s tvorbou verifikačního plánu dokáže značně zjednodušit a zrychlit tvorbu verifikací. Přístupy se mi podařilo aplikovat v rámci tvorby samotných verifikací komponent. Myslím si, že tento přístup je správný a přínosný, ale u menších komponent může být tvorba verifikačního plánu zbytečná.

U implementace samotných verifikací komponent se mi podařila ověřit jejich funkcionalita pomocí využití přístupů funkcionálních verifikací. Verifikované komponenty jsou reálně používané ve sdružení CESNET v rámci vývoje vysokorychlostních síťových karet. Proto má tato práce nejen akademický, ale také praktický smysl. Při implementaci samotných verifikací se ověřila znovupoužitelnost již vytvořených verifikací. Když už jsem měl první verifikaci implementovanou, tak mi stačilo pouze převzít strukturu projektu a založit na ni nový projekt. V novém projektu se následně použijí jiné UVM komponenty, které vyhovují specifikaci ověřované komponenty. Následně stačí vše správně propojit a vytvořit model, který korektně implementuje funkcionalitu komponenty.

Také se mi podařilo navrhnout možné rozšíření systému, které by mohlo ulehčit tvorbě nových verifikací. Pomocí navržených rozšíření systému by se daly zacílit specifičtější funkcionality a usnadnit tvorbu modelů.

Literatura

[1] Accellera Organization, Inc. *SystemVerilog 3.1a Language Reference Manual.* 2004. Dostupné z: http://courses.eees.dei.unibo.it/LABMPHSENG/wp-content/uploads/2016/02/SystemVerilog_3.1a.pdf.

[2] Accellera Systems Initiative (Accellera). Universal Verification Methodology (UVM) 1.2 User's Guide. 2015.

[3] CESNET z.s.p.o.. *ASFIFOX.* Dostupné z: https://ndk.gitlab.liberouter.org:5051/ofm/comp/base/fifo/asfifox/readme.html#asfifox.

[4] CESNET z.s.p.o.. *METADATA_EXTRACTOR: OFM private repository.* Dostupné z: https://gitlab.liberouter.org/ndk/ofm/-/blob/devel/comp/mfb_tools/flow/metadata_extractor/metadata_extractor.vhd.

[5] CESNET z.s.p.o.. *MFB_SPLITTER_SIMPLE_GEN: OFM private repository.* Dostupné z: https://gitlab.liberouter.org/ndk/ofm/-/blob/devel/comp/mfb_tools/flow/splitter_simple/mfb_splitter_simple_gen.vhd.

[6] Height, H. *A practical guide to adopting the universal verification methodology (UVM).* 2010.

[7] IEEE Computer Society. IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017).* 2020. DOI: 10.1109/IEEESTD.2020.9195920.

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

[9] Piziali, A. *Functional verification coverage measurement and analysis.* Springer Science & Business Media, 2007.

[10] Siemens Digital Industries Software. *2020 Wilson Research Group Study: FPGA trend report.* Dostupné z: https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study.

[11] Terry Bahill, A. a Henderson, S. J. Requirements development, verification, and validation exhibited in famous failures. *Systems Engineering.* 2005. DOI: https://doi.org/10.1002/sys.20017. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.20017.