VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

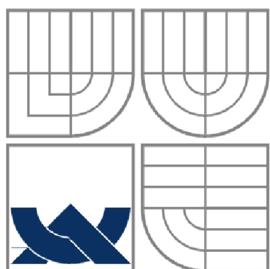
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VERIFIKACE GENERICKÉHO PROPOJOVACÍHO SYSTÉMU PRO FPGA

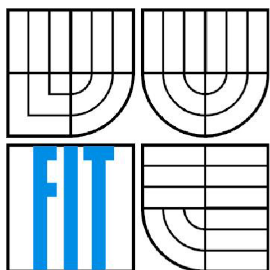
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE VÁCLAV BARTOŠ
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VERIFIKACE GENERICKÉHO PROPOJOVACÍHO SYSTÉMU PRO FPGA

VERIFICATION OF FPGA GENERIC INTERCONNECTION SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

VÁCLAV BARTOŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VIKTOR PUŠ

BRNO 2009

Abstrakt

Tato práce se zabývá návrhem, implementací a provedením simulační verifikace generického propojovacího systému pro čipy FPGA. Tento systém je součástí platformy NetCOPE vyvíjené v projektu Liberouter, v rámci něhož vznikla i tato práce.

Nejdříve jsou zde popsány obvyklé postupy návrhu verifikací v jazyce SystemVerilog. Následuje stručný popis propojovacího systému a jeho jednotlivých součástí, zaměřený především na aspekty důležité pro verifikaci. Jádrem práce je pak návrh verifikačního prostředí a řídícího programu testu pro každou ze tří součástí testovaného systému. Při tom se vychází z dříve popsaných principů zavedených v projektu Liberouter, rozšiřuje je však o některé další prvky. Všechny komponenty verifikačního prostředí jsou navrhovány s důrazem na obecnost a znovupoužitelnost, aby mohly být využity i při jiných verifikacích souvisejících s tímto propojovacím systémem.

V závěru práce jsou diskutovány výsledky provedené verifikace a nalezené chyby, a je zhodnocen obecný přínos simulačních verifikací při návrhu hardware.

Klíčová slova

Verifikace, SystemVerilog, VHDL, simulace, propojovací systém, interní sběrnice, Liberouter

Abstract

This thesis deals with design, implementation and realization of simulation verification of generic interconnection system for FPGA chips. This system is part of the NetCOPE platform developed in the Liberouter project, within which was this work done.

In the beginning, an usual methods of verification in SystemVerilog language are described. Then there is a brief description of the interconnection system, aimed especially to aspects important to verification. The main part of the thesis is design of verification environment and control program of test for all three components of the tested system. It started form the earlier described principles, that are established in the Liberouter project, and it add some more features. All components of the verification environment are designed to be general and reusable, so they can be used also in other verifications related to the interconnection system.

At the end of the thesis, there are discussed results of the verification, found bugs and the general advantages of simulation verifications.

Keywords

Verification, SystemVerilog, VHDL, simulation, interconnection system, internal bus, Liberouter

Citace

BARTOŠ, Václav: *Verifikace generického propojovacího systému pro FPGA*. Bakalářská práce. Vysoké učení technické, Fakulta Informačních technologií, Brno, 2009.

Verifikace generického propojovacího systému pro FPGA

Prohlášení

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

……………………
Václav Bartoš
19. 5. 2009

Poděkování

Především bych chtěl poděkovat svému vedoucímu bakalářské práce, panu ing. Viktoru Pušovi za mnoho užitečných rad a za čas věnovaný konzultacím. Dále bych rád poděkoval kolegům z projektu Liberouter za poskytnutí programového a technického vybavení.

© Václav Bartoš, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 3
2 Co je to verifikace 4
2.1 Řízené vs. náhodné testování 4
2.2 Pokrytí 4
2.3 SystemVerilog 5
2.4 Verifikační prostředí 6
2.4.1 Obecný princip 6
2.4.2 OVM – Open Verification Methodology 7
2.4.3 Princip verifikací v projektu Liberouter 7
2.5 Požadavky 9
3 Popis verifikovaného systému 10
3.1 Struktura 10
3.2 Rozhraní mezi komponentami 11
3.3 Komponenty 12
3.3.1 Switch 12
3.3.2 Transformer 14
3.3.3 Endpoint 15
4 Návrh verifikací 18
4.1 Obecné prvky 19
4.1.1 Transakce 19
4.1.2 Rozhraní interní sběrnice 19
4.1.3 Funkční pokrytí pro rozhraní interní sběrnice 20
4.1.4 Driver 20
4.1.5 Monitor 21
4.2 Transformer 21
4.2.1 Verifikační prostředí 21
4.2.2 Scoreboard 21
4.2.3 Funkční pokrytí 22
4.3 Switch 23
4.3.1 Verifikační prostředí 24
4.3.2 Scoreboard 24
4.3.3 Funkční pokrytí 24
4.4 Endpoint 25
4.4.1 Verifikační prostředí pro slave verzi 25
4.4.2 Uživatelské rozhraní 25
4.4.3 Funkční pokrytí uživatelského rozhraní 26
4.4.4 Transakce uživatelského rozhraní 26
4.4.5 Paměť 26
4.4.6 Scoreboard pro slave verzi 27
4.4.7 Funkční pokrytí pro slave verzi 27

4.4.8 Verifikační prostředí pro master verzi.....27
4.4.9 Bus-master rozhraní a transakce.....28
4.4.10 Bus-master monitor.....28
4.4.11 Scoreboard pro master verzi.....29
4.4.12 Funkční pokrytí pro master verzi.....29
4.5 Řízení testu a předávání parametrů.....29
5 Výsledky verifikace.....31
6 Závěr.....33
Literatura.....34
Seznam příloh.....35
Příloha A – Požadavky.....36
Příloha B – Výsledky verifikace.....40

1 Úvod

Spolu se zvyšujícími se nároky na výkon stoupá i složitost a rozsáhlost hardwarových designů. Zároveň s tím se však zvětšuje i množství chyb, které se do takového systému mohou při implementaci zanést, a jejich hledání a odstraňování je již zcela samozřejmou součástí vývoje. Čím dříve chybu odhalíme, tím méně úsilí většinou vyžaduje její odstranění. Proto jsou vyvíjené moduly obvykle před samotným nasazením v hardware testovány v simulátoru.

Tradičně se však provádí jen tzv. řízené testy, při kterých jsou vstupní data definována a výstupní kontrolována člověkem. Takovéto testy jsou dobré pro základní odladění funkčnosti, avšak pro odhalení všech chyb již často nestačí, protože v případě složitějšího systému ho člověk při testu nedokáže dostat do všech jeho možných stavů. Proto se zejména při vývoji kritických součástí systému, kde je bezchybná funkčnost za všech podmínek velmi důležitá, provádějí komplexní verifikační testy, jejichž účelem je právě otestování reakcí systému na všechny možné platné, a někdy i chybné vstupy.

Tradičně se testovací prostředí pro simulace píše ve stejném jazyce, v jakém je napsána samotná verifikovaná komponenta, tedy v Evropě obvykle ve VHDL, což je však jazyk poměrně nízké úrovně. To je výhodné pro psaní maximálně efektivních designů (z hlediska rychlosti a spotřebovaných zdrojů čipu), avšak pro simulaci je tato efektivita nepodstatná a důležitější se stává rychlost jejího vývoje. Proto je pro psaní testovacího prostředí vhodné zvolit jazyk poskytující vyšší míru abstrakce, jako je například SystemVerilog. Důležité je to především při verifikacích, kdy je nutno kvůli automatické kontrole výstupů implementovat referenční model testované komponenty a to je samozřejmě pomocí vyššího jazyka mnohem snazší.

Některé chyby mohou být způsobeny nejen chybnou implementací, ale také špatným pochopením zadání. Pokud by implementaci i verifikaci prováděl jeden člověk, pak by i do verifikačního modelu mohla být zanesena stejná chyba, jako do testované jednotky. Proto by verifikaci měl provádět vždy někdo jiný, než autor jednotky. Šance, že zadání pochopí stejně špatně dva různí lidé je minimální.

Tato práce se zabývá návrhem, implementací a provedením verifikace jednotlivých komponent generického propojovacího systému pro FPGA čipy. Tento systém je novou verzí interních sběrnic platformy NetCOPE vyvíjené v rámci projektu Liberouter[1]. Protože pomocí tohoto systému komunikují všechny ostatní jednotky na čipu, a to jak mezi sebou, tak s hostitelským systémem (zpravidla počítač), je velmi důležité aby se choval naprosto bezchybně. Je proto nutné ho před nasazením verifikovat a tak co nejvíce chyb (nejlépe všechny) odhalit a odstranit ještě ve fázi simulací.

Práce je logicky strukturována do několika částí. V následující kapitole jsou podrobně popsány principy verifikace a způsoby jejich návrhu, dále, v kapitole 3, je popsán verifikovaný systém interních sběrnic. Jádrem práce je pak návrh verifikačních prostředí a průběhu testů pro jednotlivé jeho komponenty. V předposlední kapitole jsou uvedeny výsledky provedené verifikace včetně nalezených chyb. V závěru pak jsou důležité části práce zrekapitulovány, je zhodnocen přínos provedené verifikace a možnost znovuvyužití některých navržených prvků pro další verifikační projekty.

1 Liberouter je výzkumný záměr sdružení CESNET zabývající se vývojem síťových aplikací založených na reprogramovatelných čipech FPGA. Pro více informací viz www.liberouter.org.

2 Co je to verifikace

Slovo verifikace znamená *ověřování*. Při verifikaci nějakého systému tedy ověřujeme, že neobsahuje chyby a jeho chování za všech podmínek přesně odpovídá specifikaci. V této práci se budeme zabývat verifikací hardwarového designu (přesněji, zdrojového kódu, ze kterého může být syntetizován číslicový obvod) a to *verifikací pomocí simulace*.

Existuje totiž ještě jiná metoda, a to *formální verifikace*. Ta k ověřování využívá matematické formální metody, například temporální logiku a metodu *model checking*. Lze pomocí ní správnost nebo nesprávnost implementace určité vlastnosti daného systému dokázat, je ale velice složitá a časově náročná [1].

Simulační metoda verifikace k ověření využívá velké množství experimentů prováděných v simulačním software. Sice pomocí ní nelze dokázat bezchybnost, je ale mnohem jednodušší a rychlejší na implementaci. Navíc pokud je dobře navržena a provedena, je pravděpodobnost nenalezení nějaké existující chyby poměrně malá.

2.1 Řízené vs. náhodné testování

Simulace hardwarových komponent i celých systémů je běžná. Vytvoří se virtuální testovací prostředí, tzv. *testbench*, do něj se testovaná komponenta zapojí a na její vstupy jsou posílána různá data. Obvykle však jde o jednoduché *řízené testy*, což znamená, že množina vstupních dat je předem definována vývojářem. Správnost výstupů se v tomto případě také obvykle kontroluje „ručně" sledováním hodnot jednotlivých výstupních signálů. Toto je vhodné pro ověření základní funkčnosti za běžných podmínek, nebo pro testování velmi jednoduchých jednotek s malým počtem vnitřních stavů.

Při verifikaci ovšem potřebujeme ověřit, zda systém reaguje správně za všech podmínek na všechna, i když třeba velmi nepravděpodobná, vstupní data. Dosáhnout všech legálních kombinací vstupních dat a dostat verifikovaný systém do všech možných vnitřních stavů by při použití řízených testů bylo, zvláště pro složitější systémy, velmi náročné a zdlouhavé.

Proto se pro verifikace používá metoda *constrained random testing* (což by se dalo přeložit jako *omezeně náhodné testování*). Tato metoda spočívá v tom, že vstupy testovaného systému jsou generovány náhodně[1] v rámci definovaných mezí. Ty omezují vstupní hodnoty především tak, aby odpovídaly protokolu rozhraní verifikovaného systému. Můžeme vstupy omezit i více, pokud například chceme zvlášť otestovat různé krajní stavy. Výstupy systému jsou kontrolovány automaticky, což nám spolu s automatickým generováním vstupů umožňuje snadno provádět i velmi dlouhé simulační běhy, při nichž se otestuje velké množství stavů systému.

2.2 Pokrytí

Abychom však měli přehled o tom, jak dobře je již systém otestován, musíme generované vstupy a případně i výstupy sledovat a vyhodnocovat, které vlastnosti systému testovány byly a které ještě ne. Míře již otestovaných vlastností budeme říkat *funkční pokrytí* (v anglické literatuře *functional*

1 Ve skutečnosti pseudonáhodně, aby byly výsledky simulací reprodukovatelné.

coverage). Musíme však definovat, které hodnoty, skupiny hodnot, kombinace hodnot různých veličin, případně jejich posloupnosti, jsou pro test důležité a budeme jejich výskyt sledovat.

Mějme například sběrnici, kde rozlišujeme 4 typy transakcí a délku přenášených dat. Můžeme například definovat, že z délek jsou podstatné jen tři skupiny hodnot, nejkratší možná délka, nejdelší a cokoliv mezi tím. Dále určíme, že každá z těchto tří délek se musí vyskytnout se všemi typy transakcí. Cílem verifikace pak bude dosažení toho, aby se při testu vyskytlo všech 12 definovaných kombinací – tedy dosažení 100% pokrytí. V tomto případě bychom toho dosáhli snadněji jednoduchým řízeným testem, je to však jen příklad, ve skutečnosti bývá sledovaných hodnot mnohem více a počet kombinací tak jde do tisíců.

Správná volba hodnot, které budeme pro funkční pokrytí sledovat, je velmi důležitá. Nejjistějším řešením by bylo požadovat naprosto všechny legální kombinace a posloupnosti všech hodnot. Těch však může být u složitých systémů obrovské množství a bylo by nutné provádět neúměrně dlouhé simulace. Pokud však víme, že pro určitou skupinu hodnot se testovaný systém musí chovat vždy stejně, stačí nám otestovat jen jednu hodnotu z této skupiny a množství požadovaných kombinací tak výrazně snížit (jako jsme to udělali ve výše uvedeném příkladě s délkou transakcí).

Kromě správného určení množiny sledovaných hodnot každé veličiny musíme také určit, které veličiny na sobě závisí a je tedy nutné otestovat všechny jejich vzájemné kombinace, a které jsou nezávislé a stačí pokrýt jejich množinu hodnot bez ohledu na veličiny ostatní.

Kromě funkčního pokrytí se sledují i další ukazatele kompletnosti verifikace, především *pokrytí kódu* (*code coverage*). To se dělí na *pokrytí příkazů* (*statement coverage*), které nám říká, které příkazy ve zdrojovém kódu byly alespoň jednou vykonány a které ne, *pokrytí podmínek* (*condition coverage*), sledující jestli byla každá podmínka vyhodnocena jako pravda i jako nepravda, a několik dalších. Pokrytí kódu sleduje sám simulační software, takže není nutné nic definovat jako u pokrytí funkčního.

I zde se samozřejmě snažíme dosáhnout 100% pokrytí, pokud se nám to nedaří, znamená to buď, že nedostatečně testujeme, nebo že je v designu nějaký kus kódu zbytečný, protože se nikdy nemůže provést.

2.3 SystemVerilog

Vzhledem k tomu, že testovací prostředí musí automaticky kontrolovat správnost reakce systému na různé vstupy, musí v něm být implementován stejný algoritmus jako ve verifikované jednotce. Aby toto nebylo příliš složité, mělo by být možno tento algoritmus zapsat pomocí vyššího programovacího jazyka. Zároveň musí být použitý jazyk vhodný pro simulaci číslicových obvodů. Tyto a i další požadavky nejlépe splňuje jazyk *SystemVerilog*. [1, kap. 4]

SystemVerilog je poměrně nové (standardizováno v roce 2005 jako IEEE P1800-2005) a rozsáhlé rozšíření jazyka Verilog, jenž je často používaným jazykem pro návrh hardware (*HDL – hardware description language*). SystemVerilog ho rozšiřuje jak o další prvky pro návrh, tak především o prvky převzaté z jazyků pro verifikaci (*HVL – hardware verifiacation language*, například *OpenVera* nebo *e*). Vytvořil tak novou kategorii programovacích jazyků, zvanou *HDVL – hardware description and verification language* (jazyk pro návrh a verifikaci hardwaru) [2, 3].

Tento jazyk má mnoho vlastností, které jsou výhodné pro provádění verifikací, jako například propracovaný systém generování náhodných hodnot, prostředky pro definici rozhraní a jednoduché propojování jednotek, nebo tzv. assertions, které lze využít pro automatickou kontrolu komunikač-

ního protokolu. Dále podporuje objektově orientované programování, dynamická vlákna a meziprocesovou komunikaci.

Navíc k němu lze připojit i moduly psané v jiném jazyce, a tak lze simulovat i designy psané například v SystemC nebo VHDL. Toto je pro nás velmi důležité, protože systém interních sběrnic, jehož verifikací se tato práce zabývá, je napsán právě ve VHDL.

Posledním, avšak možná nejdůležitějším důvodem použití SystemVerilogu pro tuto práci je skutečnost, že v projektu Liberouter se tento jazyk pro verifikace již používá, a tak lze využít zde zavedené principy a některé hotové obecné jednotky.

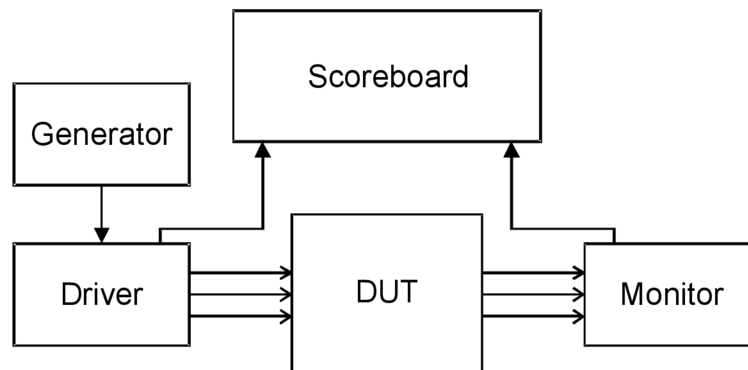
2.4 Verifikační prostředí

Pro provedení verifikace je nutno v simulátoru zapojit kromě verifikované komponenty také další prvky, které generují transakce, sledují výstupy a vyhodnocují výsledky. Systému těchto prvků se říká *verifikační prostředí*.

Existují různé metodologie, jak ho vytvářet , my si zde nejdříve ukážeme základní princip jak toto prostředí obvykle vypadá (alespoň při použití jazyka SystemVerilog).

2.4.1 Obecný princip

Základní verifikační prostředí pro jednoduchou komponentu s jedním vstupním a jedním výstupním rozhraním je znázorněno na obrázku 2.1. Uprostřed vidíme verifikovanou jednotku (*DUT – design under test*), jejíž vstupy i výstupy jsou propojeny s testovacím prostředím. To se skládá z několika základních prvků: *generátor*, *driver*, *monitor* a *scoreboard*.



Obr. 2.1: Schéma jednoduchého verifikačního prostředí

Hlavní datovou strukturou, se kterou při verifikacích pracujeme je *transakce*. Můžeme ji definovat jako sled logicky souvisejících signálů nebo jako základní jednotku komunikačního protokolu. Může jí tedy být například jedna instrukce procesoru, příkaz nebo paket. Ve verifikačním prostředí jsou transakce reprezentovány strukturou nebo objektem SystemVerilogu.

V generátoru jsou transakce v rámci definovaných omezení náhodně vytvářeny a poté odesílány do komponenty driver. Jejím úkolem je převádět tyto abstraktní transakce na konkrétní signály vstupního rozhraní testované jednotky. Navíc při odeslání každé transakce je její kopie předána i komponentě scoreboard.

Monitor sleduje hodnoty signálů na výstupním rozhraní verifikované jednotky, převádí je na transakce reprezentované opět objektem a ty předává do scoreboardu.

Ten je vlastně vyhodnocovací jednotkou, která určuje, zda zachycené výstupy odpovídají vygenerovaným vstupům. Musí v něm tedy být implementován stejný algoritmus, jaký provádí verifikovaná komponenta. Při přijetí transakce od driveru scoreboard pomocí tohoto algoritmu rozhodne, co se má s transakcí stát, vygeneruje očekávaný výstup a uloží ho do tzv. *tabulky transakcí*. Skutečný výstup je později zachycen monitorem a jako transakce je předán scoreboardu. Ten se ji pokusí najít v tabulce a pokud se tam přesně taková vyskytuje, vymaže ji z ní. V případě, že se však přijatou transakci v tabulce najít nepodaří, je hlášena chyba, protože to znamená, že se na výstupním rozhraní objevila transakce, která nebyla očekávána. Navíc po skončení simulace musí být tabulka transakcí prázdná, pokud není, znamená to, že ne všechny transakce očekávané na výstupu se tam skutečně objevily, a je opět hlášena chyba.

Při verifikaci komponenty s více vstupními a výstupními rozhraními je princip obdobný. Komponenty driver a monitor se připojí na každé vstupní, resp. výstupní rozhraní a scoreboard bude obsahovat více tabulek transakcí (zpravidla jednu pro každé výstupní rozhraní).

2.4.2 OVM – Open Verification Methodology

Jednou z velmi používaných verifikačních metod je *Open verification methodology* (*otevřená metodika pro verifikace*), která je výsledkem společného vývoje dvou velkých výrobců simulačních nástrojů – Cadence a Mentor Graphics. Nejde jen o metodologii, ale především o standardní knihovnu základních tříd, umožňující vytvoření modulárního a znovupoužitelného verifikačního prostředí, v němž komponenty komunikují pomocí standardních rozhraní. Tato knihovna je dostupná pod licencí Apache 2.0, která umožňuje její využití zdarma komukoliv a k jakýmkoliv účelů.

Výhodné vlastnosti této metodologie zahrnují komunikaci komponent na úrovni transakcí, která je základem modulárnost a znovupoužitelnost, definici jednotlivých fází běhu simulace (vytvoření komponent, jejich propojení, nastavení, ...), jež je důležitá pro správný běh mnoha různých komponent z různých zdrojů v jednom prostředí, dále možnost měnit verifikační prostředí za běhu a vytvářet více testů v jednom základním prostředí jen s minimálními změnami v kódu, univerzální způsob předávání parametrů mezi komponentami hierarchicky shora dolů, a další. [4]

S touto metodologií je navíc kompatibilní mnoho prodávaných verifikačních komponent sloužících obvykle ke zpracovávání standardních protokolů [5, 6]. Tato metodologie je tak vhodná pro použití ve velkých firmách, kde se provádí mnoho verifikací různých rozsáhlých systémů, používajících však často stejná rozhraní. Pro účely této práce je ale tato metodologie zbytečně robustní a příliš složitá a proto ji používat nebudeme.

2.4.3 Princip verifikací v projektu Liberouter

V projektu Liberouter, v rámci něhož tato práce vznikla, je zavedeno jednodušší, avšak dostačující schéma verifikací. To zde bude popsáno podrobněji, protože z něj budeme vycházet při návrhu verifikačního prostředí pro komponenty interní sběrnice v kapitole 4.

Každý prvek verifikačního prostředí je představován třídou SystemVerilogu. Tyto prvky jsou zastřešovány modulem `Test`, ve kterém jsou vytvořeny, propojeny a nastaveny, a který obsahuje hlavní testovací smyčku, která řídí simulaci.

Generování transakcí
Jako generátor je používána vždy stejná univerzální třída `generator`, která je schopna pracovat s jakýmkoliv typem transakcí. To je možné díky tomu, že každá třída popisující transakce je zděděna z obecné třídy `Transaction` obsahující virtuální funkce `display`, `copy` a `compare`.

Generování pak probíhá tak, že v hlavním modulu je vytvořena vzorová transakce obsahující definici omezení náhodných proměnných a ta je předána generátoru. Ten ji ve smyčce neustále kopíruje, kopiím nechává vygenerovat nové hodnoty náhodných proměnných a vkládá je do schránky (`mailbox`[1]), ze které si je vyčítá driver.

Rozhraní
Dále je nutno definovat rozhraní testované jednotky. Zde se kromě názvů a šířek signálů a jejich směru specifikují i některá pravidla komunikačního protokolu. Ty jsou popisována pomocí temporální logiky v konstrukcích `sequence` a `assertion`, a tak jsou během simulace neustále automaticky kontrolována.

Driver a monitor
Když máme definovanou podobu transakcí i rozhraní, můžeme vytvořit třídy pro driver a monitor. Aby byly tyto třídy univerzální a bylo je možno použít pro různé testy a pro libovolnou jednotku s daným rozhraním, používají systém tzv. *callback* funkcí (popsaný také v [2, kap. 8.7]). Ten spočívá v tom, že předávání transakcí do scoreboardu, jejich změna před samotným odesláním a další akce specifické pro konkrétní test, nejsou definovány přímo v driveru (resp. monitoru). Ten místo toho obsahuje seznam funkcí, které volá bezprostředně před a po odeslání transakce. Tyto funkce jsou jednoduše přidávány z hlavního modulu testu a tak může být chování driveru a monitoru rozšiřováno, aniž by bylo nutné jakkoliv zasahovat přímo do jejich kódu.

Seznam callback funkcí je ve skutečnosti seznam objektů – takových objetků, jejichž třída dědí z obecné třídy `DriverCbs` (resp. `MonitorCbs`) funkce `pre_tx` a `post_tx` (resp. `pre_rx` a `post_rx`) a alespoň jednu z těchto funkcí implementuje.

Mnoho protokolů rozhraní umožňuje do komunikace vkládat čekací stavy, to pak obvykle zajišťuje také driver a monitor. Generování těchto stavů bývá náhodné, nastavitelné několika proměnnými přístupnými z hlavního modulu testu.

Scoreboard
Předchozí prvky se musí definovat pro každé rozhraní, scoreboard je naproti tomu specifický pro každou verifikovanou jednotku. Jsou v něm obvykle definovány třídy s callback funkcemi sloužícími k přidávání nebo odebírání transakcí do nebo z tabulek. Tyto tabulky jsou také instancovány zde. Vzhledem k tomu, že všechny transakce jsou zděděny z jedné obecné třídy, lze pro tabulku transakcí používat také jen jen jednu univerzální třídu. V té je implementována dynamická paměť pro libovolné transakce, funkce pro jejich přidávání a funkce pro jejich vyhledávání a mazání.

---

1 `Mailbox` je konstrukce jazyka SystemVerilog sloužící k předávání dat mezi procesy. Můžeme si ji představit jako paměť typu FIFO, do které jeden proces zapisuje a druhý z ní čte.

Verifikovaná jednotka
Hardwarové komponenty jsou na projektu Liberouter většinou psány v jazyce VHDL. Pro testované jednotky se proto vytváří obálka sloužící k převodu konkrétních signálů jejího rozhraní na abstraktní rozhraní v SystemVerilogu.

Test
Hlavní program v modulu `Test` obvykle nejdříve vytvoří všechny objekty a propojí je (to spočívá například v předávání ukazatelů na rozhraní konstruktorům objektů nebo registraci callback funkcí). Poté verifikovanou jednotku resetuje a spustí generátory. Čeká, až systémem projde předem zadané množství transakcí, a poté test ukončí. Takových testů může být v jedné simulaci i více za sebou, přičemž se mění některé parametry simulace (například rozložení typů posílaných transakcí nebo generování čekacích stavů driverem a monitorem).

2.5 Požadavky

Před samotným návrhem verifikačního prostředí a jednotlivých jeho prvků je nutné jasně specifikovat, jak se má verifikovaný systém správně chovat. To se obvykle definuje jako sada jednoduchých výroků, které musí být pravdivé, aby se o daném systému dalo říci, že pracuje správně. Každý z těchto výroků pak musí být ve verifikačním prostředí nějakým způsobem testován. Tato sada výroků se obvykle označuje anglickým termínem *requirements*, česky tedy požadavky či podmínky.

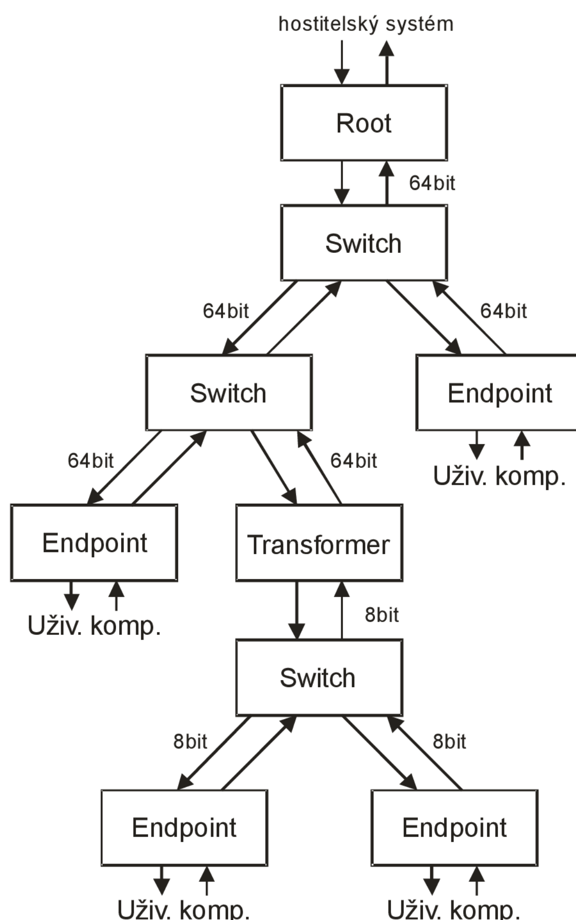
Je důležité, aby zde byly popsány skutečně všechny aspekty chování systému. Pokud na něco zapomeneme, nebude tato funkčnost testována a může nám tak nějaká chyba uniknout.

3 Popis verifikovaného systému

Tato kapitola stručně popisuje systém interních sběrnic pro FPGA čipy, jehož verifikace je předmětem této práce. Tento popis je velice zjednodušený, zaměřený především na aspekty důležité pro verifikaci, tedy chování jednotlivých částí systému navenek, nikoliv jejich vnitřní implementaci. Informace v této kapitole jsou čerpány z diplomové práce Tomáše Málka [7] zabývající se návrhem tohoto systému.

3.1 Struktura

Generický propojovací systém (GICS – generic interconnection system) je sběrnice, sloužící jak ke komunikaci mezi různými komponentami na FPGA čipu, tak k jejich komunikaci s hostitelským systémem. Na obrázku 3.1 je znázorněna jeho stromová struktura. Nahoře vidíme *kořenovou komponentu (root)*, která zajišťuje komunikaci s hostitelským systémem. Větvení stromové struktury zajišťuje *přepínací komponenta (switch)*, která má vždy jeden port vedoucí směrem ke kořeni (*upstream* směr) a dva porty směrem k listům *(downstream* směr). Listy stromu jsou představovány *koncovými komponentami (endpoint)*, pomocí nichž se ke sběrnici připojují různé uživatelské jednotky.



Obr. 3.1: Příklad zapojení propojovacího systému

Protože jednotlivé uživatelské komponenty mají různé požadavky na rychlost přenosu, mohou mít různé části sběrnice různou datovou šířku. K propojení takových částí a převodu jedné datové šířky na druhou, slouží *transformační komponenta (transformer)*. Šířka sběrnice může být libovolná mocnina dvou v rozsahu 8 až 128 bitů.

Dále v textu budou komponenty sběrnice nazývány jejich anglickými názvy.

3.2 Rozhraní mezi komponentami

Jednotlivé komponenty sběrnice jsou vždy propojeny dvěma linkami, jednou pro každý směr. Přenos dat na sběrnici je tak plně duplexní. Každá linka sestává z pěti signálů – datového, jenž má nastavitelnou šířku, a čtyř řídících. Jejich význam je uveden v tabulce 3.1. Řídící signály používají negativní logiku. Toto uspořádání linky odpovídá nejjednodušší variantě rozhraní LocalLink [8] firmy Xilinx.

název	směr	šířka	popis
DATA	zdroj → cíl	8–128	Data
SOF_N	zdroj → cíl	1	Začátek paketu
EOF_N	zdroj → cíl	1	Konec paketu
SRC_RDY_N	zdroj → cíl	1	Zdroj je připraven vysílat (data jsou platná)
DST_RDY_N	cíl → zdroj	1	Cíl je připraven přijímat

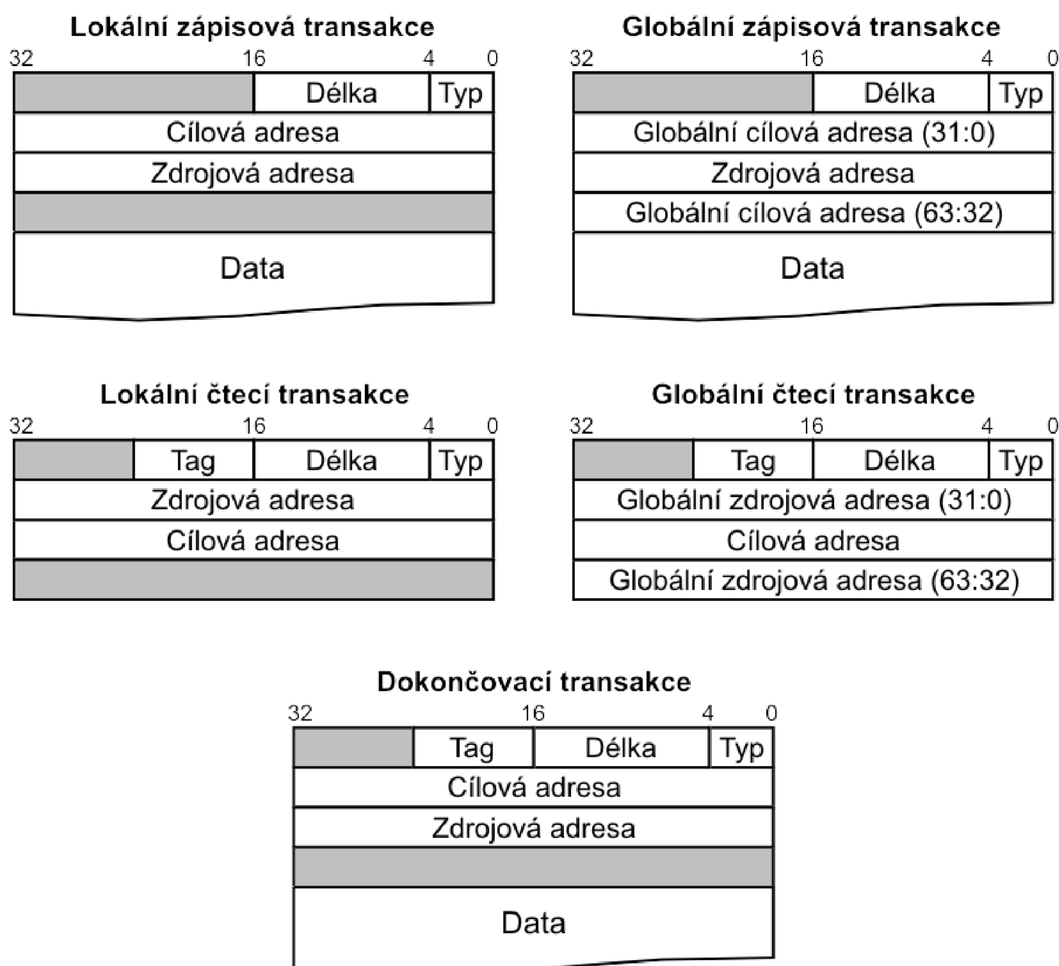
Tabulka 3.1: Signály rozhraní interní sběrnice

Na sběrnici se může vyskytnout 5 typů transakcí. Zápisová, čtecí (požadavek na čtení) a dokončovací (odpověď, přečtená data), z nichž první dvě mohou být buď lokální, tedy v rámci čipu, nebo globální, tedy směřující z čipu do hostitelského systému. Transakce iniciované hostitelským systémem nejsou označovány jako globální, protože kořenová komponenta je vždy překládá do formátu lokální transakce.

Transakce jsou na sběrnici reprezentovány pakety. Každý paket se skládá ze 128bitové hlavičky a v případně zápisových a dokončovacích transakcí až 4 kB dat. Hlavička obsahuje informaci o typu paketu, délce přenášených dat a zdrojovou a cílovou adresu. Čtecí a dokončovací transakce navíc obsahují identifikační číslo (*tag*), pomocí něhož se vzájemně párují. Formáty paketů jednotlivých transakcí jsou znázorněny na obrázku 3.2.

Data jsou v paketech vždy zarovnána na cílovou adresu, to znamená, že první bajt není vždy umístěn na začátku prvního slova, ale všechna data mohou být o několik bajtů posunuta. Tento posun je dán n nejnižšími bity cílové adresy paketu, kde n je rovno dvojkovému logaritmu datové šířky sběrnice v bajtech. Budeme-li mít například šířku 64 bitů a nenižší 3 bity adresy paketu budou mít hodnotu 5, budou v prvním datovém slově transakce platné jen nejvyšší tři bajty a spodních 5 zůstane nevyužitých. Další nevyužité bajty mohou být v posledním slově, kde je jejich počet dán součtem tohoto posunutí a celkové délky dat. Transakcím, jejichž cílová adresa má nejnižších n bitů nenulových, se říká *nezarovnané transakce*.

Dále platí, že transakce nesmí tzv. křižovat 4kB stránku paměti, což znamená, že součet délky dat a nejnižších 12ti bitů adresy nesmí být větší než 4096.



Obr. 3.2: Formát paketů interní sběrnice (pro datovou šířku 32 bitů)

3.3 Komponenty

V této podkapitole si podrobněji popíšeme jednotlivé komponenty switch, transformer a endpoint, jejichž verifikací se tato práce zabývá. Kromě popisu jejich funkčnosti se zaměříme především na všechny nastavitelné parametry. Každá komponenta totiž lze genericky nastavovat a množství různých variant nám bude později značně komplikovat verifikaci (především u endpointu).

3.3.1 Switch

Switch má tři porty interní sběrnice, všechny se stejnou datovou šířkou. Jeden port směrem ke kořeni stromové struktury sběrnice (upstream) a dva porty směrem k listům (downstream). Jeho jedinou funkcí je směrování transakcí – rozpoznává cílovou adresu paketů, které přijdou na nějaký port, a přeposílá je na jeden ze dvou zbývajících portů. Rozsahy adres jednotlivých portů jsou, stejně jako datová šířka, nastavitelné pomocí generiků[1].

Switch má dvě varianty, *master* a *slave*, které se liší způsobem směrování paketů. Master verze vybírá port, na který má být paket směrován, podle následujícího algoritmu (doslova převzato z [7]):

1 *Generik* je konstrukce jazyka VHDL umožňující parametrizaci jednotlivých instancí určité komponenty. Pojem pochází z klíčového slova `generic`, jímž se tato konstrukce vytváří.

- Při příchodu paketu z downstream portu:
  - Pokud cílová adresa paketu spadá do rozsahu sousedního downstream portu, pak je transakce směrována na tento port.
  - Pokud cílová adresa paketu spadá do rozsahu upstream portu nebo se jedná o globální transakci nesoucí adresu globálního prostoru, pak je transakce směrována na upstream port.
  - V jiných případech je paket zahozen. Nemůže být směrován na neexistující adresu ani na stejný port ze kterého přišel.
- Při příchodu paketu z upstream portu:
  - Pokud cílová adresa paketu spadá do rozsahu některého z downstream portů, pak je transakce směrována na tento port.
  - V jiných případech je paket zahozen. Nemůže být směrován na neexistující adresu ani na stejný port, ze kterého přišel.

Slave varianta používá mnohem jednodušší algoritmus:

- Při příchodu paketu z downstream portu je transakce automaticky směrována na upstream port.
- Při příchodu paketu z upstram portu je transakce směrována oba downstream porty.

Tato jednodušší verze sice zabírá mnohem méně zdrojů čipu, ale přináší s sebou několik omezení. Protože jsou všechny transakce příchozí zdola směrovány nahoru, není možné, aby mezi sebou komunikovaly komponenty zapojené do různých podstromů vytvářených tímto switchem. Navíc je třeba, aby endpointy zapojené pod tento switch obsahovaly adresový dekodér, protože pakety jsou směrovány na oba downstream porty a tedy přijdou i k endpointu, do jehož adresového prostoru nepatří.

V obou variantách switche lze nastavit velikost vstupních vyrovnávacích pamětí (datové buffery). Ty se využijí v případě, že na port, kam má být transakce přeposlána, je právě posílána transakce jiná. Tehdy musí druhá transakce čekat, a pokud by na vstupu nebyla dostatečně velká vyrovnávací paměť, blokovala by se tak část sběrnice, ze které transakce přišla.

Přehled všech nastavitelných parametrů je shrnut v tabulce 3.2. Parametry nastavující adresové prostory platí jen pro master verzi.

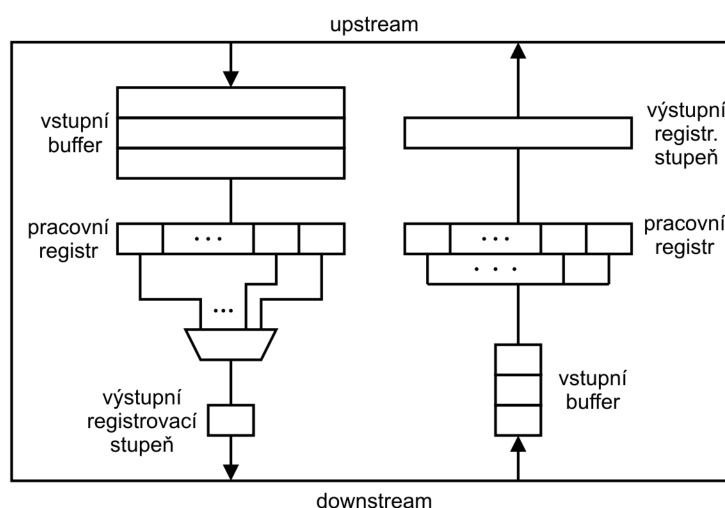
Název parametru	Možné hodnoty	Popis
DATA_WIDTH	8, 16, 32, 64, 128	Datová šířka
HEADER_NUM	1 – n	Velikost vstupní vyrovnávací paměti
SWITCH_BASE	0 – 0xFFFFFFFF	Spodní hranice adresového prostoru switche
SWITCH_LIMIT	0 – 0xFFFFFFFF	Velikost adresového prostoru switche
DOWN1_BASE	0 – 0xFFFFFFFF	Spodní hranice adresového prostoru portu 1
DOWN1_LIMIT	0 – 0xFFFFFFFF	Velikost adresového prostoru portu 1
DOWN2_BASE	0 – 0xFFFFFFFF	Spodní hranice adresového prostoru portu 2
DOWN2_LIMIT	0 – 0xFFFFFFFF	Velikost adresového prostoru portu 2

Tabulka 3.2: Nastavitelné parametry komponenty switch

3.3.2 Transformer

Transformační komponenta slouží k převodu jedné datové šířky na druhou. Má tedy dvě rozhraní, každé s jinou datovou šířkou. Transakce žádným způsobem neupravuje, pouze pakety přeposílá z jednoho rozhraní na druhé.

Transformer data při převodu z vyšší datové šířky rozhraní na nižší jednoduše multiplexuje, při převodu opačném je zase postupně ukládá do registru a poté odesílá celé slovo vyšší datové šířky (viz obrázek 3.3). Pokud je však cílová adresa nebo součet adresy a délky nezarovnaný, jsou některé bajty slova na vyšší datové šířce nevyužity. To musí transformer řešit a při převodu na nižší datovou šířku takové bajty ignorovat, při převodu na vyšší je naopak doplňovat. Proto musí sledovat cílovou adresu a délku transakcí, jiné jejich parametry jsou pro něj nepodstatné.



Obr. 3.3: Vnitřní architektura transformeru [7]

I transformer má na obou rozhraních vyrovnávací paměti. Ty jsou zde důležité proto, že na různých datových šířkách nemohou data téci stejnou rychlostí, a bez bufferů by byla rychlejší sběrnice zbytečně blokována. Na obou výstupech může být pro zlepšení časování zapnut registrovací stupeň. Přehled všech nastavitelných parametrů je uveden v tabulce 3.3.

Název parametru	Možné hodnoty	Popis
UP_DATA_WIDTH	16, 32, 64, 128	Datová šířka horního rozhraní
DOWN_DATA_WIDTH	8, 16, 32, 64	Datová šířka dolního rozhraní
UP_INPUT_BUFFER_ITEMS	0 – n	Velikost vstupní vyrovnávací paměti na horním rozhraní
DOWN_INPUT_BUFFER_ITEMS	0 – n	Velikost vstupní vyrovnávací paměti na dolním rozhraní
UP_OUTPUT_PIPE	true / false	Výstupní registrovací stupeň na horním rozhraní
DOWN_OUTPUT_PIPE	true / false	Výstupní registrovací stupeň na dolním rozhraní

Tabulka 3.3: Nastavitelné parametry komponenty transformer

3.3.3 Endpoint

Endpoint tvoří list stromové struktury sběrnice, pomocí něj se ke sběrnici připojují různé jednotky. Má tedy na jedné straně vstupní a výstupní rozhraní interní sběrnice, na druhé straně obecné zápisové a čtecí rozhraní a volitelně rozhraní pro generování transakcí uživatelskou komponentou (tzv. *bus-master*).

Nastavitelné parametry

Podle přítomnosti bus-master rozhraní rozlišujeme dvě základní varianty endpointu – *master* a *slave*. Jednodušší slave verze umožňuje pouze přijímat zápisové transakce a čtecí požadavky a na tyto požadavky odpovídat transakcemi dokončovacími. Master verze obsahuje navíc bus-master rozhraní a dovoluje tak uživatelské komponentě čtecí a zápisové transakce generovat.

Kromě tohoto dělení má endpoint i mnoho jiných nastavitelných parametů (generiků). Samozřejmostí je nastavitelná datová šířka, stejně jako u ostatních komponent interní sběrnice. Lze nastavit i šířku adresy, která je předávána uživatelské komponentě. Ta je dána velikostí adresového prostoru této komponenty. Dále je možno pomocí generiku nastavit, že je endpoint připojen ke slave verzi switche, a tak zapojit adresový dekodér. V tom případě je nutné definovat ještě konkrétní rozsah adresového prostoru.

Čtecí rozhraní endpointu může pracovat v jednom ze dvou módů – *kontinuální* a *paketový*. Při kontinuálním čtení se data žádají po jednotlivých slovech z postupně rostoucí adresy. Při paketovém čtení je nastavena jen počáteční adresa a množství požadovaných dat. Požadavek je tak odbaven v jediném taktu a hned za ním můžou následovat další čtecí nebo zápisové transakce.

Než se tedy vrátí všechna požadovaná data, může přijít několik dalších čtecích požadavků. Aby endpoint mohl vygenerovat správné dokončovací transakce, musí si všechny nedokončené požadavky pamatovat. Velikost paměti, do které si tyto požadavky ukládá, je také nastavitelná.

Vzhledem k tomu, že při nezarovnaných transakcích jsou data v paketech vždy umístěna podle cílové adresy, je často nutné data přečtená z uživatelské komponenty přerovnat. Pokud však zajistíme, že všechny příchozí čtecí transakce mají adresu zarovnánu, nebude to potřeba. Proto lze přerovnávací obvod genericky zapnout nebo vypnout.

Posledním nastavitelným parametrem je velikost vstupní a výstupní vyrovnávací paměti. Přehled všech parametrů je uveden v tabulce 3.4.

Název parametru	Možné hodnoty	Popis
DATA_WIDTH	8, 16, 32, 64, 128	Datová šířka
ADDR_WIDTH	1 – 32	Šířka adresy na uživatelském rozhraní
BUS_MASTER_ENABLE	true / false	Přítomnost bus-master rozhraní
ENDPOINT_BASE	0 – 0xFFFFFFFF	Spodní hranice adresového prostoru
ENDPOINT_LIMIT	0 – 0xFFFFFFFF	Velikost adresového prostoru
CONNECTED_TO	SWITCH_MASTER, SWITCH_SLAVE	Typ switche, ke kterému je endpoint připojen (určuje přítomnost adresového dekodéru)
DATA_REORDER	true / false	Přítomnost přerovnávacího obvodu
READ_TYPE	CONTINUAL, PACKET	Typ protokolu čtecího rozhraní
READ_IN_PROCESS	1 – n	Maximum nedokončených čtecích operací v jednom okamžiku (velikost paměti na hlavičky čtecích transakcí)
INPUT_BUFFER_SIZE	0 – n	Velikost vstupní vyrovnávací paměti
OUTPUT_BUFFER_SIZE	0 – n	Velikost výstupní vyrovnávací paměti

Tabulka 3.4: Nastavitelné parametry endpointu

Rozhraní uživatelské komponenty

Signály zápisového a čtecího rozhraní jsou popsány v tabulkách 3.5 a 3.6. Bus-master rozhraní je téměř stejné jako rozhraní interní sběrnice. Pokud chce uživatelská jednotka vygenerovat transakci, pošle na rozhraní hlavičku paketu této transakce. V případě čtecí transakce je hlavička beze změny odeslána na sběrnici, v případě transakce zápisové jsou nejdříve přes čtecí rozhraní z uživatelské komponenty přečtena zapisovaná data a teprve poté je celá transakce odeslána na sběrnici.

Oproti interní sběrnici má bus-master rozhraní navíc ještě signály `TAG` a `TAG_VLD`. Ty indikují dokončení vygenerované transakce – zapsání posledního přečteného bajtu do uživatelské komponenty v případě čtecí transakce, odeslání posledního bajtu na sběrnici v případě zápisové. Transakce jsou identifikovány číslem z položky `TAG` v hlavičce paketu. Toto číslo je pak při dokončení transakce obsahem signálu `TAG`, signál `TAG_VLD` označuje jeho platnost.

Název signálu	Směr	Šířka	Popis
WR_ADDR	E → U	1 – 32	Adresa, kam mají být data zapsána.
WR_DATA	E → U	8 – 128	Data
WR_BE	E → U	1 – 16	Povolovací signály (byte enables). Jednotlivé bity tohoto signálu označují platnost odpovídajících bajtů dat. Jejich šířka je dána šířkou dat vydělenou osmi.
WR_REQ	E → U	1	Požadavek (request). Označuje platnost všech ostatních signálů.
WR_RDY	U → E	1	Připraven (ready). Označuje připravenost uživatelské komponenty přijmout data. K přenosu dochází pouze tehdy, jsou-li oba signály WR_REQ i WR_RDY aktivní.
WR_LENGTH	E → U	12	Délka (length). Počet bajtů dat, která budou zapsána v rámci této transakce.
WR_SOF	E → U	1	Začátek transakce (start of frame).
WR_EOF	E → U	1	Konec transakce (end of frame).

Tabulka 3.5: Signály zápisového rozhraní endpointu

Název signálu	Směr	Šířka	Popis
RD_ADDR	E → U	1 – 32	Adresa, odkud mají být data přečtena
RD_BE	E → U	1 – 16	Povolovací signály (byte enables). Platné pouze při kontinuálním čtení. Jednotlivé bity tohoto signálu označují, jestli mají být přečteny odpovídající bajty dat. Jejich šířka je dána šířkou dat vydělenou osmi.
RD_REQ	E → U	1	Požadavek (request), označuje platnost všech ostatních signálů.
RD_ARDY_ACCEPT	U → E	1	Připraven (address ready, accept). Označuje připravenost uživatelské komponenty zpracovat požadavek. K přenosu požadavku a čtení dat dochází pouze tehdy, jsou-li oba signály RD_REQ i RD_ARDY_ACCEPT aktivní.
RD_LENGTH	E → U	12	Délka (length). Počet bajtů dat, která mají být přečtena. Důležité především při paketovém čtení.
RD_SOF_IN	E → U	1	Začátek transakce (start of frame).
RD_EOF_IN	E → U	1	Konec transakce (end of frame).
RD_DATA	U → E	8 – 128	Přečtená data.
RD_SRC_RDY	U → E	1	Source ready. Značí, že uživatelská komponenta přečetla data a chce je odeslat (tj. signál RD_DATA je platný)
RD_DST_RDY	E → U	1	Destination ready. Značí že endpoint je schopen přijmout data. K přenosu dochází pouze tehdy, jsou-li oba signály RD_SRC_RDY i RD_DST_RDY aktivní.

Tabulka 3.6: Signály čtecího rozhraní endpointu

4 Návrh verifikací

Hlavní náplní této práce je návrh a implementace verifikačního prostředí pro jednotlivé komponenty propojovacího systému. Budeme při tom vycházet z principů popsaných výše, včetně použití jazyka SystemVerilog.

Před samotným návrhem je však nutné jasně specifikovat, které vlastnosti systému budeme testovat a přesně jaké chování budeme očekávat – tedy tzv. požadavky. Byl sepsán jejich seznam pro každou z verifikovaných komponent a pro všechna použitá rozhraní. Aby se minimalizovala pravděpodobnost, že bude tato specifikace chybná nebo nedostatečná, byly tyto požadavky psány ve spolupráci s dalšími lidmi, kteří se v tomto propojovacím systému vyznají. Jejich přesné znění, podle kterého budeme postupovat při návrhu, je uvedeno v příloze A.

Hlavní problém při návrhu verifikačních prostředí bude představovat množství nastavitelných parametrů testovaných komponent. Zejména u endpointu lze vytvořit mnoho variant s různým chováním. Verifikační prostředí tedy musí být také nastavitelné, aby generované transakce i kontrola výstupů odpovídaly aktuální konfiguraci komponenty. Při verifikaci takových komponent je žádoucí otestovat všechny jejich možné varianty. To je v našem případě možné u komponent switch a transformer, jejichž počet parametrů není příliš vysoký. U transformeru jsou nastavitelné jen datové šířky a velikosti vyrovnávacích pamětí, u switche kromě toho ještě dvě varianty chování. Počty možných kombinací tak jdou maximálně do desítek.

Endpoint má však parametrů více a množství jejich kombinací roste exponenciálně. Budeme-li u velikosti vyrovnávacích pamětí a dalších parametrů, jejichž hodnotou může být libovolné číslo, počítat s testováním jen třech až čtyř různých hodnot, vychází nám celkový počet možných kombinací několik desítek tisíc! Verifikace jedné kombinace může i na dnešních výkonných počítačích trvat několik hodin, proto je provedení takového množství simulací nereálné. Avšak ne všechny parametry mohou ovlivnit všechny součásti endpointu, například přítomnost výstupního bufferu s největší pravděpodobností nemá žádný vliv na funkčnost adresového dekodéru na vstupu. Proto by k nalezení všech chyb mělo být dostačující otestovat jen několik desítek vhodně vybraných kombinací.

I tak však bude nutné provést nemalé množství simulací s různým nastavením, proto je třeba, aby šly všechny potřebné parametry nastavovat snadno a na jednom místě.

Kromě verifikace jednotlivých komponent je do budoucna plánována i verifikace propojovacího systému jako celku, pravděpodobně i s dalšími, k němu připojenými, komponentami. Na to je vhodné pamatovat již nyní a navrhnout prvky verifikačního prostředí tak univerzálně, aby se mohly beze změny použít i v takovéto komplexnější verifikaci. Navíc je nutné tomu přizpůsobit i způsob předávání parametrů, protože v takové verifikaci bude více instancí jednotlivých komponent a u každé instance můžeme vyžadovat jiné nastavení.

Než se však budeme zabývat těmito detaily, musíme navrhnout základní strukturu verifikačních prostředí.

4.1 Obecné prvky

Každá komponenta systému interních sběrnic sice bude verifikována zvlášť, a bude tak nutné navrhnout tři různá verifikační prostředí, všechny však používají stejné rozhraní, takže mnoho tříd těchto prostředí bude stejných pro všechny tři komponenty (a pro jakoukoliv jinou verifikaci ve které se objeví rozhraní interní sběrnice). Nejdříve tedy navrhneme tyto obecné prvky.

4.1.1 Transakce

Nejdříve si definujeme třídu popisující transakce. Ta musí obsahovat takové proměnné, aby kompletně popisovaly paket interní sběrnice. Bude to tedy typ, lokální a globální adresa, délka, tag a data. Tyto proměnné se budou pro každou transakci generovat náhodně. Ne všechny kombinace jsou však legální (například některé typy transakcí nenesou data), proto musí být náhodné generování omezeno pomocí tzv. `constraints`. Tato omezení musí zahrnovat následující:

- Čtecí transakce neobsahují data
- Délka dat transakce je minimálně 1 B a maximálně 4096 B
- Součet spodních 12ti bitů adresy a délky dat nesmí být větší než 4096

Dále je žádoucí, aby bylo možno z vyšších úrovní testu nastavovat další omezení. Konkrétní hodnoty těchto omezení budou nastavitelné pomocí několika proměnných přístupných zvnějšku. Ty budou zahrnovat:

- Pravděpodobnosti jednotlivých typů transakcí
- Dolní a horní hranici délky transakce
- Dolní a horní hranici tagu
- Dolní a horní hranici lokální adresy
- Dolní a horní hranici globální adresy
- Počet bitů zarovnání adres (počet spodních bitů adresy, které musí být nulové)

Protože tato třída bude zděděna z obecné třídy `Transaction`, musí implementovat v ní deklarované funkce – `copy`, která vrací kompletní kopii transakce, `compare` porovnávájící ji s jinou transakcí, a `display` vypisující obsah transakce na standardní výstup.

4.1.2 Rozhraní interní sběrnice

Dále je nutné vytvořit popis používaného rozhraní. Z důvodu použití prvku `clocking`, který usnadňuje správné časování zajištěním toho, že hodnoty signálů jsou skrz něj čteny vždy těsně před hodinovým signálem a zapisovány vždy těsně po něm, a proto, že tento prvek bude vždy použit na straně testovacího prostředí, je nutné rozlišit dva typy rozhraní – jedno vedoucí z testovacího prostředí do verifikované komponenty a druhé naopak.

Obě tato rozhraní budou obsahovat definici signálů interní sběrnice – `DATA`, `SOF_N`, `EOF_N`, `SRC_RDY_N` a `DST_RDY_N`. Budou se lišit jen směrem těchto signálů. Zároveň budou v obou definována `assertions`[1] kontrolující některá pravidla protokolu sběrnice. Podle požadavků uvedených v příloze A určíme, že je nutno kontrolovat pravdivost těchto výroků:

1 Tvrzení, která musí během simulace platit, jinak je ohlášena chyba.

- Pokud datová šířka není 128 bitů, SOF_N a EOF_N nesmí být aktivní zároveň.
- Po EOF_N nesmí být aktivní SRC_RDY_N, dokud není nepřijde SOF_N.
- Každý SOF_N musí být po nějaké době následován EOF_N.

Vzhledem k tomu, že velká část popisu obou rozhraní bude stejná, bylo by vhodné využít dědičnost, aby se zbytečně neopakoval stejný kód. SystemVerilog však u definic rozhraní dědičnost nepodporuje.

4.1.3 Funkční pokrytí pro rozhraní interní sběrnice

Jak bylo popsáno výše, při verifikaci je nutné sledovat, jaké vstupy z definované množiny už byly do jednotky poslány a jaké ještě ne. Na úrovni rozhraní budeme vyžadovat, aby se objevily určité kombinace hodnot řídících signálů, případně jejich posloupnosti. Pro rozhraní interní sběrnice definujeme takovéto množiny hodnot:

- Všechny kombinace signálů SOF_N, SRC_RDY_N a DST_RDY_N.
- Všechny kombinace signálů EOF_N, SRC_RDY_N a DST_RDY_N.
- Všechny tříprvkové posloupnosti kombinací signálů SRC_RDY_N a DST_RDY_N.

Žádná z takových kombinací neporušuje protokol sběrnice, ale některé nastat nemusí. U prvních dvou množin jsou to kombinace hodnot, kdy je SOF_N aktivní, ale SRC_RDY_N neaktivní, u poslední jsou to takové posloupnosti, kdy je SRC_RDY_N aktivní, ale DST_RDY_N neaktivní a v následujícím taktu se SRC_RDY_N deaktivuje (přestože nedošlo k přenosu dat). Takové chování sice neporušuje protokol, ale od testovaných jednotek se neočekává, proto jejich výskyt na výstupních rozhraních (z pohledu testované jednotky) nebude vyžadován. Na vstupní rozhraní je však odesílat budeme. Datový signál sledován není, pokrytí různých vlastností transakcí bude řešeno později.

4.1.4 Driver

Driver interní sběrnice je třída, která přebírá transakce od generátoru a odesílá je na rozhraní interní sběrnice. Ukazatel na objekt tohoto rozhraní je předáván v konstruktoru, stejně jako ukazatel na schránku ze které se transakce budou odebírat.

Driver obsahuje metody pro spuštění a zastavení automatického odesílání. Pokud je aktivní, ve smyčce neustále vybírá transakce ze schránky (pokud tam jsou, tedy pokud generátor ještě nějaké vytváří) a odesílá je v podobě posloupností hodnot signálů rozhraní do verifikované komponenty. Kromě toho je vždy možné odeslat nějakou konkrétní transakci „ručním“ voláním metody pro odesílání.

Do scoreboardu jsou transakce předávány pomocí systému callback funkcí, tak jak to bylo popsáno v kapitole 2.4.3 – před odesláním transakce nebo hned poté jsou volány všechny dříve zaregistrované funkce. Těm je aktuální transakce předána jako parametr a mohou ji libovolně zpracovávat, funkce volané před odesláním i měnit.

Při odesílání transakcí driver náhodně nastavuje signál SRC_RDY_N a generuje tak čekací stavy. Pravděpodobnost vložení čekacího stavu a jeho minimální a maximální délka jsou nastavitelné pomocí několika proměnných. Navíc je rozlišováno čekání mezi transakcemi a uvnitř transakcí.

4.1.5 Monitor

Monitor interní sběrnice je třída, která zachytává transakce přicházející na rozhraní interní sběrnice a pomocí callback funkcí je předává k dalšímu zpracování (typicky do scoreboardu).

Podobně jako u driveru je mu ukazatel na rozhraní předán v konstruktoru a také obsahuje metody pro spuštění a zastavení. Pokud je aktivní, sleduje signály na interní sběrnici, převádí je na objekty reprezentující transakce a volá zaregistrované callback funkce, jimž tyto transakce předává.

Zároveň náhodně nastavuje signál `DST_RDY_N` a generuje tak čekací stavy. Stejně jako u driveru lze pravděpodobnost jejich vložení a nejmenší a největší délku nastavit pomocí několika proměnných, a to opět různě pro čekání mezi transakcemi a uvnitř transakcí.

Všechny tyto třídy, kromě popisu transakce, jsou parametrizovatelné, s jediným parametrem určujícím datovou šířku rozhraní.

Nyní tedy máme navrženy všechny společné prvky a můžeme je využít při sestavování verifikačních prostředí pro jednotlivé komponenty interní sběrnice.

4.2 Transformer

Začneme návrhem prostředí pro komponentu transformer, protože ta je z hlediska verifikace nejjednodušší.

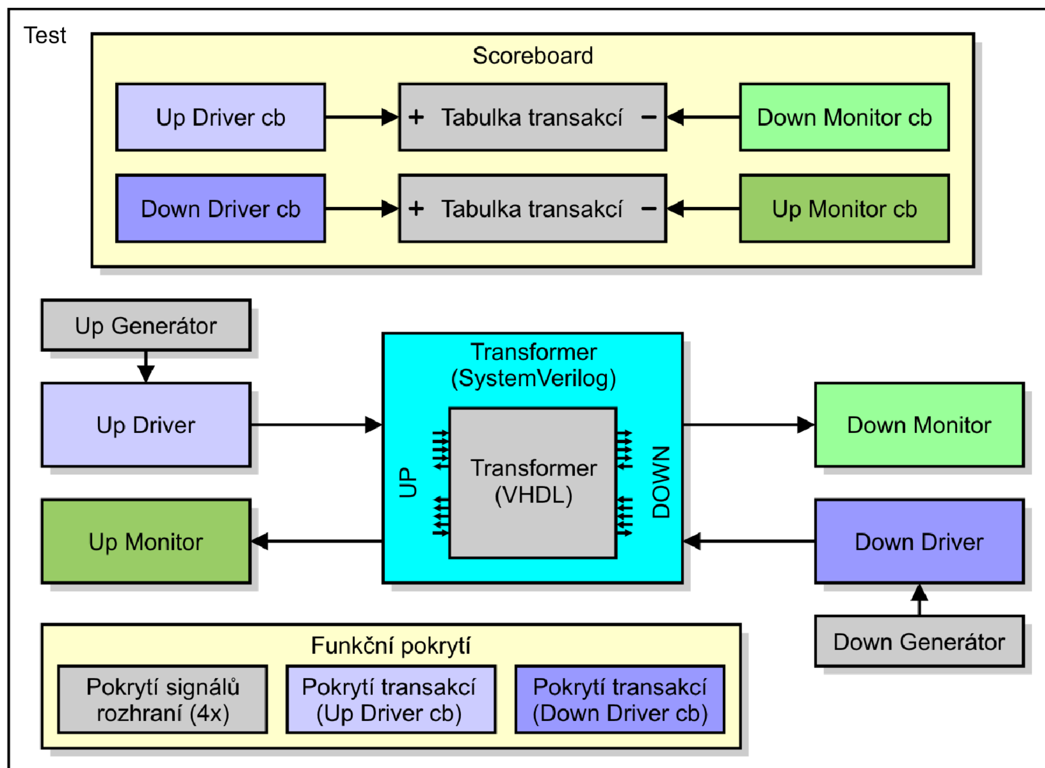
4.2.1 Verifikační prostředí

Transformer má jen dvě vstupní a dvě výstupní rozhraní, každá vstupně-výstupní dvojice má jinou datovou šířku. Je kolem něj, stejně jako kolem každé jiné verifikované komponenty, vytvořena obálka převádějící jednotlivé skupiny signálů na rozhraní SystemVerilogu. K oběma vstupním rozhraním zapojíme driver, k výstupním monitor. Každému driveru je navíc přiřazena jedna instance univerzálního generátoru.

Každé verifikační prostředí musí obsahovat ještě vyhodnocovací jednotku (*scoreboard*) a je zastřešováno a řízeno nějakým hlavním modulem (*test*). Celé navržené prostředí je znázorněno na obrázku 4.1.

4.2.2 Scoreboard

Ve scoreboardu jsou dvě tabulky (jedna pro každý směr), do kterých se ukládají transakce předpokládané na výstupech transformeru. Pro předávání vstupních a skutečných výstupních transakcí je zde definováno zde několik callback funkcí (ve schématu označeny *cb*), které jsou na začátku simulace zaregistrovány v driverech a monitorech. Ve funkcích náležejících driveru bývá obvykle implementován algoritmus odpovídající funkčnosti verifikované komponenty, pomocí něhož vygeneruje očekávaný výstup. Avšak vzhledem k tomu, že před odesláním driverem a po přijetí monitorem je transakce reprezentována objektem, který je nezávislý na datové šířce, a jinak transformer transakce nemění, nemusí zde v tomto případě být implementován algoritmus žádný. Transakce jsou jen převzaty od driveru a beze změny uloženy do tabulky (ukládání je ve schématu označeno šipkou a znaménkem plus).



Obr. 4.1: Schéma verifikačního prostředí pro transformer

Callback funkce registrované v monitorech od nich přebírají transakce, které se objevily na výstupních rozhraních a snaží se je vyhledat v příslušné tabulce. Pokud tam taková transakce je, znamená to, že se na výstupu objevila právě taková transakce, jaká byla očekávána. Vymaže se tedy z tabulky a pokračuje se dál. Pokud se však v tabulce přijatá transakce nenalezne, je hlášena chyba.

Ve skutečnosti se tabulka neprohledává celá, ale porovnávání se provádí jen s nejstarší transakcí v tabulce. Tím je zajištěna kontrola, že transformer odesílá transakce ve stejném pořadí, v jakém je přijal.

4.2.3 Funkční pokrytí

Jak bylo uvedeno v popisu transformeru (kapitola 3.3.2), je pro něj podstatné zarovnání dat v paketech, tedy cílová adresa, délka transakcí a jejich typ (protože čtecí transakce data nenesou). Budeme proto měřit pokrytí těchto vlastností transakcí, abychom zajistili, že do něj byly poslány transakce se všemi možnými zarovnáními, a otestovali jsme ho tedy úplně.

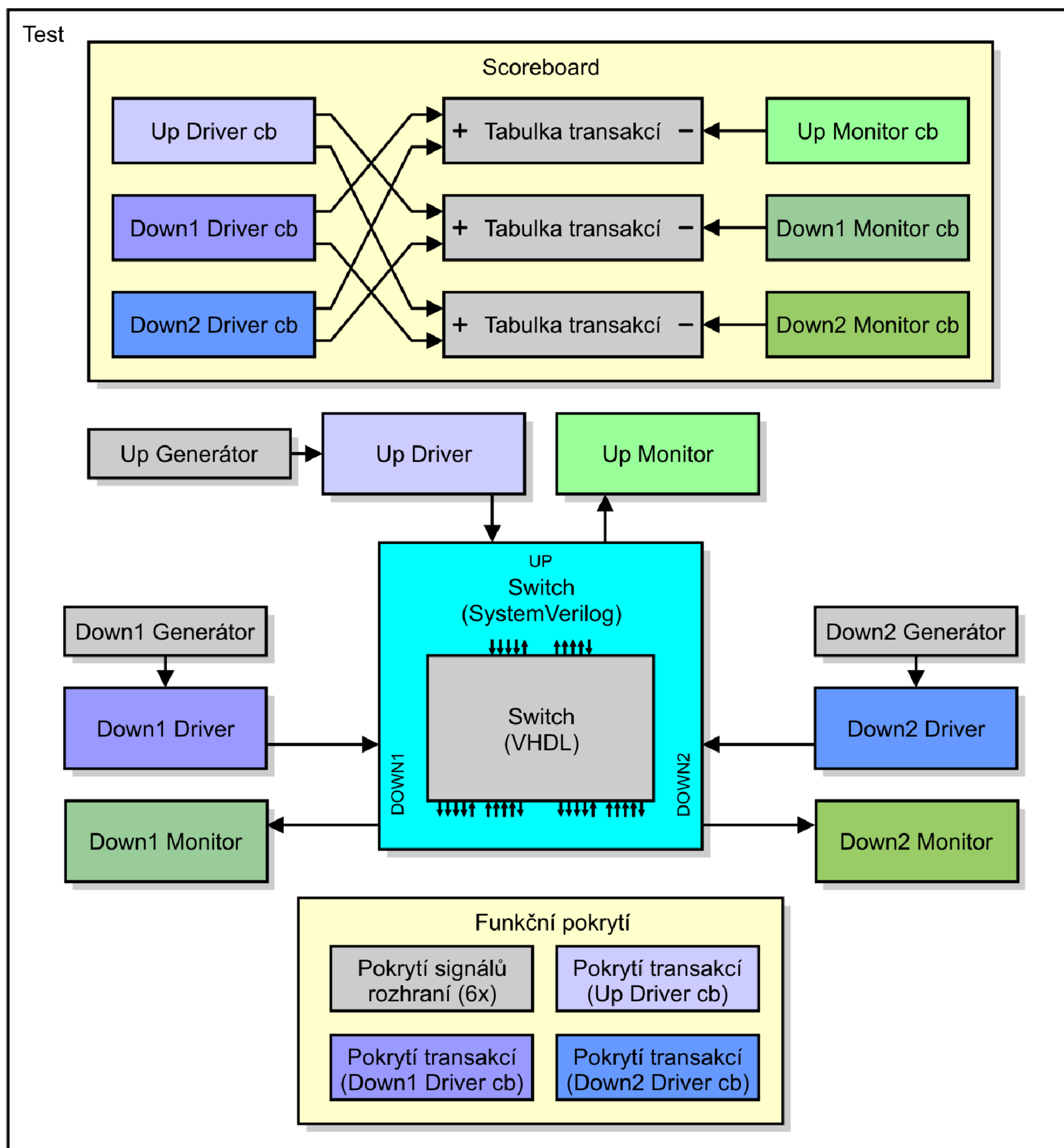
Vytvoříme třídu obsahující konstrukce pro sledování pokrytí všech kombinací typu transakce, n nejnižších bitů cílové adresy a n nejnižších bitů součtu cílové adresy a délky (n závisí na datové šířce rozhraní). Transakce sem budou posílány pomocí callback funkce, kterou v této třídě vytvoříme, a která se při spuštění simulace zaregistruje v driveru (budou celkem dvě instance této třídy, jedna pro každý driver).

Dále je měřeno pokrytí kombinací signálů na každém rozhraní, a to pomocí čtyř objektů třídy popsané v kapitole 4.1.3.

Princip funkce hlavního modulu testu, způsob nastavování parametrů, průběh verifikace a podmínky jejího automatického ukončení jsou obdobné pro všechny tři komponenty a budou proto popsány až dále, v kapitole 4.5.

4.3 Switch

Verifikace switche je o něco složitější, transakce sice také žádným způsobem nemění, ale je v něm implementován směrovací algoritmus. Ve scoreboardu tedy musíme také rozhodovat, na který port se má transakce přeposlat.



Obr. 4.2: Schéma verifikačního prostředí pro switch

4.3.1 Verifikační prostředí

Navržené verifikační prostředí pro switch je znázorněno na obrázku 4.2. Kolem verifikované komponenty je opět vytvořena obálka převádějící skupiny signálů na abstraktní rozhraní a opět je na každé vstupní rozhraní připojen driver, na výstupní monitor. Každému driveru je přiřazen vlastní generátor transakcí. Potud je, až na počet rozhraní, vše stejné jako u transformeru, jiný je především scoreboard a také vlastnosti transakcí, které se budou sledovat pro funkční pokrytí.

4.3.2 Scoreboard

Ve scoreboardu jsou tentokrát tři tabulky, každá obsahuje transakce předpokládané na jednom z výstupních rozhraní. Do těchto tabulek jsou transakce opět vkládány pomocí callback funkcí, zaregistrovaných v driverech. Tyto funkce musí implementovat stejný směrovací algoritmus, jaký je ve switchi. Při příchodu transakce je tedy porovnána její cílová adresa s nastavenými konstantami určujícími adresové prostory jednotlivých portů a je rozhodnuto, do které z tabulek bude transakce uložena, případně jestli má být zahozena a tedy do žádné tabulky neukládána. Toto platí pro master verzi switche.

Pokud má switch nastavenu slave variantu chování, musí se změnit i chování funkcí ve scoreboardu. Callback funkce driveru na horním rozhraní pak vkládá transakce do tabulek obou dolních rozhraní, ostatní funkce je vkládají do tabulky rozhraní horního.

Z tabulek jsou transakce vyjímány pomocí callback funkcí zaregistrovaných v monitorech. Přijatá transakce musí samozřejmě opět přesně odpovídat transakci vyjímané z tabulky, jinak je hlášena chyba.

4.3.3 Funkční pokrytí

Sledování pokrytí transakcí zde bude probíhat stejně jako u verifikace transformeru, tedy pomocí callback funkcí, kterými budou transakce předávány třídám s definovanými množinami vlastností transakcí. Jen tyto vlastnosti budou jiné.

Jedinými položkami paketu, které mají pro switch význam, jsou cílová adresa a typ paketu (konkrétně jen bit, určující, zda je transakce lokální či globální). Budeme proto požadovat, aby se při testu na všech portech vyskytly pakety se všemi důležitými adresami (požadovat výskyt úplně všech adres je vzhledem jejich množství nemyslitelné) a to pro oba typy transakcí. Množiny adres, jejichž pokrytí budeme požadovat jsou navrženy takto: Musí se vyskytnout alespoň jedna transakce s cílovou adresou ...

- v adresovém prostoru každého portu.
- mimo jakýkoliv adresový prostor.
- rovnající se všem hranicím adresových prostorů.
- lišící se od hranic adresových prostorů o ± 8, 16, 32, 64, 128, 256, 65536 bajtů.
- lišící se od hranic adresových prostorů a od předchozího o ± 1.

Navíc bude nutné spustit test několikrát s různým nastavením adresových prostorů.

Kromě pokrytí adres transakcí bude opět měřeno i pokrytí posloupností hodnot řídících signálů na všech rozhraních. Slave verze switche se na adresu paketů vůbec nedívá a při její verifikaci tedy stačí měřit jen pokrytí signálů rozhraní.

4.4 Endpoint

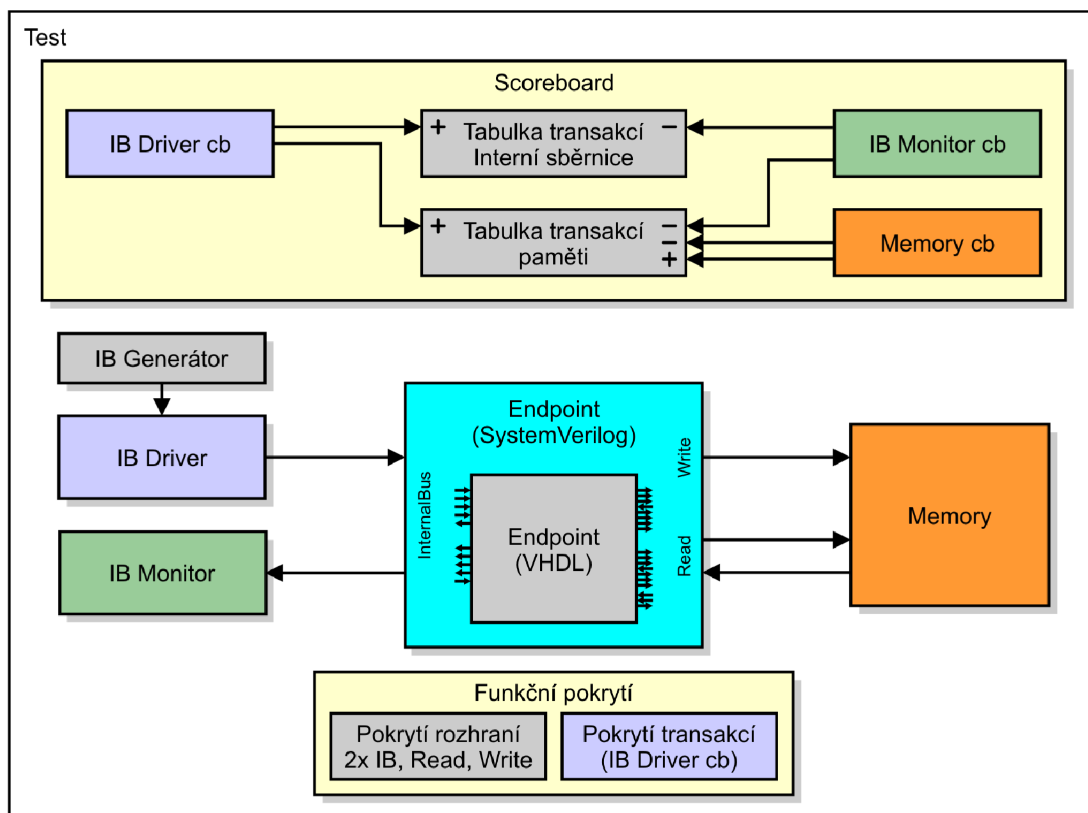
Verifikace endpointu je ze všech tří komponent nejsložitější. Kromě toho, že samotná jeho funkčnost je složitější než u ostatních komponent, navíc obsahuje více typů rozhraní a je nutno simulovat i uživatelskou komponentu, která odpovídá na čtecí požadavky. Kromě scoreboardu, měření funkčního pokrytí a celkového zapojení, zde musíme navíc navrhnout definici uživatelského rozhraní, třídu pro transakce posílané skrz něj a jednotku simulující jednoduchou paměť připojitelnou k tomuto rozhraní. Pro master verzi ještě prvky pro práci s bus-master rozhraním.

4.4.1 Verifikační prostředí pro slave verzi

Nejdříve navrhneme verifikační prostředí pro jednodušší variantu endpointu bez bus-master rozhraní (viz obrázek 4.3). Uprostřed vidíme endpoint, jako vždy uzavřený v obálce v SystemVerilogu. Na rozhraní interní sběrnice je připojen driver a monitor. Na druhé straně, na uživatelském rozhraní, je připojen modul simulující paměť. Než navrhneme tento modul, popíšeme jeho rozhraní a transakce.

4.4.2 Uživatelské rozhraní

Rozhraní endpointu pro připojení uživatelské komponenty má dvě části – zápisovou a čtecí (v případě master varianty endpointu ještě bus-master, ale tou se budeme zabývat později). Tyto části budeme implementovat jako dvě nezávislá rozhraní. U obou definujeme všechny signály a jejich směr. Stejně jako u rozhraní interní sběrnice i zde použijeme na straně testovacího prostředí konstrukci `clocking` řešící správné časování při čtení a zápisu hodnot signálů.



Obr. 4.3: Schéma verifikačního prostředí pro slave variantu endpointu

Opět zde budeme pomocí assertions kontrolovat některé prvky komunikačního protokolu. Při jejich definici vyjdeme z požadavků uvedených v příloze A. Na obou rozhraních budeme kontrolovat splnění následujících pravidel:

- Signál `BE` nikdy nesmí obsahovat samé nuly (pokud je aktivní `REQ`).
- Uprostřed transakce (je aktivní `REQ`, ale ne `SOF` ani `EOF`) musí `BE` obsahovat samé jedničky.
- `REQ` nesmí být aktivní mezi `EOF` a `SOF`.
- Signál `SOF` nesmí být aktivní dvakrát po sobě, aniž by mezitím byl aktivní `EOF`.

4.4.3 Funkční pokrytí uživatelského rozhraní

Podobně jako sledujeme výskyt různých posloupností hodnot na rozhraní interní sběrnice, budeme takové posloupnosti sledovat i zde. Na zápisovém rozhraní půjde o kombinaci všech možných tří-prvkových posloupností signálů `REQ` a `RDY`. Na rozhraní čtecím to bude podobně pro dvojici signálů `REQ` a `ARDY_ACCEPT` a navíc ještě dvojici `SRC_RDY` a `DST_RDY`.

Nejsou to však vždy zcela všechny kombinace, i zde je nutné odebrat některé, které nedávají smysl. Je při tom nutno rozlišovat rozdílné chování při nastavení kontinuálního nebo paketového typu čtení.

4.4.4 Transakce uživatelského rozhraní

Transakce na zápisovém i čtecím rozhraní budeme reprezentovat jednou třídou. Ta bude obsahovat proměnné určující typ transakce (zápis, čtecí požadavek nebo odpověď), její adresu, délku a samotná přenášená data. Tyto transakce nebudou nikde náhodně generovány, a proto nemusí být definována žádná omezení jejich hodnot, jako v případě transakcí interní sběrnice.

4.4.5 Paměť

Paměť, jednotka simulující chování jednoduché uživatelské komponenty připojené k endpointu, v sobě zahrnuje monitor a driver zároveň. Na zápisovém rozhraní přijímá data a ukládá je na příslušnou adresu do paměti. Kontroluje přitom i některé prvky komunikačního protokolu, například jestli množství přenesených dat odpovídá hodnotě signálu `LENGTH`. Na čtecím rozhraní přijímá požadavky, přečte požadovaná data z paměti a vloží je do softwarové obdoby posuvného registru, která slouží k simulaci latence paměti. Délka latence je nastavitelná. Po několika taktech, když data tímto registrem projdou, jsou odeslána do endpointu. Každý zápis dat, příjem čtecího požadavku a odeslání odpovědi má navíc za následek vygenerování transakce, která reprezentuje tuto operaci, a která je předána dále pomocí příslušných callback funkcí.

Podobně jako u driveru a monitoru interní sběrnice, i zde dochází k vkládání čekacích stavů do komuniakce, a to pomocí náhodného nastavování signálů `WR_RDY` na zápisovém rozhraní, `RD_ARDY_ACCEPT` při přijímání požadavku na čtecím rozhraní a `SRC_RDY` při odesílání požadovaných dat. Četnost těchto čekacích stavů i jejich minimální a maximální délku lze opět nastavovat pomocí několika proměnných.

Třída reprezentující paměť je navíc parametrizovatelná – při jejím vytváření je nutno určit šířku datových a adresových signálů a také typ čtecích požadavků (paketové nebo kontinuální).

4.4.6 Scoreboard pro slave verzi

Nyní, když máme navrženy všechny potřebné jednotky, můžeme si popsat scoreboard, čímž si vysvětlíme fungování celého verifikačního prostředí.

Jsou zde dvě tabulky, jedna pro transakce interní sběrnice, druhá pro transakce uživatelského rozhraní. Dále je zde opět několik callback funkcí, ty jsou však o něco složitější, než v předchozích případech. Funkce zaregistrovaná v driveru podle přijímaných transakcí interní sběrnice vytváří odpovídající transakce uživatelského rozhraní a ukládá je do příslušné tabulky (viz šipky na obr. 4.3). V případě čtecích transakcí vygeneruje ještě hlavičku očekávané dokončovací transakce a uloží ji do tabulky interní sběrnice. Pouze hlavičku generuje proto, že nemůže znát data, která budou přečtena. Později je porovnávána opět jen hlavička.

Třída s callback funkcemi paměti obsahuje tyto funkce tři. Jsou volány při příchodu zápisové transakce, čtecího požadavku, nebo při odeslání požadovaných dat. V prvních dvou případech jsou přijaté transakce z tabulky vyjímány, ve třetím jsou tam vkládány.

Funkce zaregistrovaná v monitoru interní sběrnice přijímá dokončovací transakce, jejich hlavičky vyhledává v tabulce interní sběrnice a odstraňuje je z ní. Data a adresu těchto transakcí převádí na transakce uživatelského rozhraní a ty vyhledává a vyjímá z druhé tabulky. Tak je kontrolována správnost hlavičky i dat dokončovacích transakcí přesto, že driver při odesílání požadavku neví, jaká data mají být vrácena, a paměť zase nezná všechny položky hlavičky.

4.4.7 Funkční pokrytí pro slave verzi

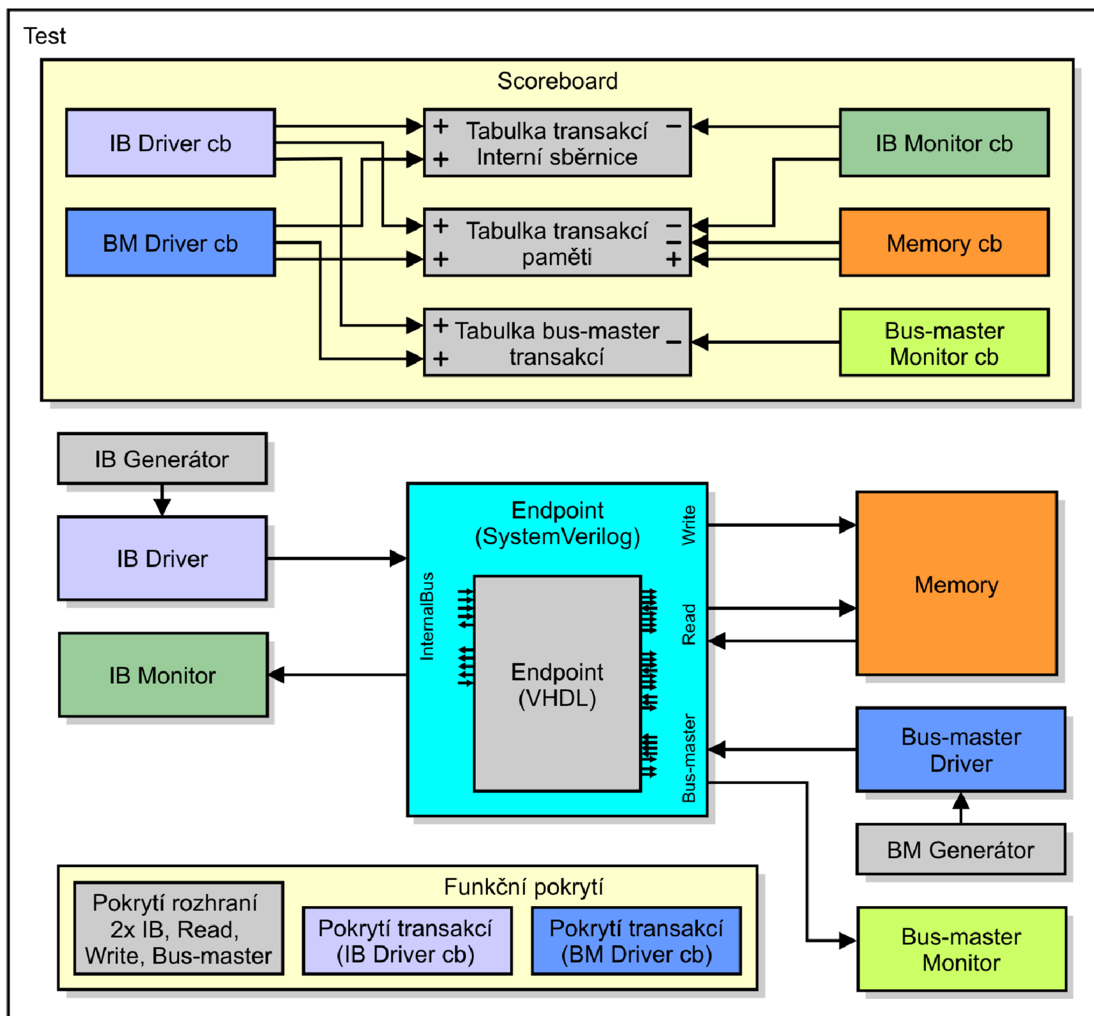
Abychom v průběhu testu znali míru otestování endpointu, bude sledováno pokrytí kombinací signálů na interní sběrnici i uživatelském rozhraní. Především se však budou sledovat téměř všechny parametry generovaných transakcí. Pro prohlášení stoprocentního pokrytí, tedy toho, že byly plně otestovány všechny funkce endpointu, musí do něj být poslány transakce se všemi těmito parametry:

- Všechny délky dat od 1 bajtu po počtu bajtů daného dvojnásobkem datové šířky, délky 4095 a 4096 bajtů (maximum) a jakákoliv délka mezi tím. To vše navíc zvlášť pro oba typy transakcí (čtecí nebo zápisové).
- Pokud je zapnut parametr endpointu `DATA_REORDER`, pak pro oba typy transakcí všechna zarovnání cílové adresy – tj. všechny kombinace nejnižších *n* bitů adresy, kde $n = \log_2$(datová šířka v bajtech).
- Pokud je zapnut parametr endpointu `DATA_REORDER`, pak pro oba typy transakcí všechna zarovnání součtu cílové adresy a délky.

4.4.8 Verifikační prostředí pro master verzi

Při návrhu prostředí pro verifikaci master verze endpointu si popíšeme jen rozdíly proti slave verzi. Hlavní změnou je přítomnost bus-master rozhraní. Na něj budeme transakce posílat pomocí driveru interní sběrnice a generátoru. Ten však vytváří transakce i s daty, ale na bus-master rozhraní se mají posílat jen hlavičky. Abychom nemuseli zasahovat do obecných komponent, vyřešíme tento problém pomocí callback funkce. Ta bude driverem volána před odesláním každé transakce a odstraní z ní všechna data, aby zůstala jen hlavička.

Další změnou je to, že generátor transakcí interní sběrnice bude vytvářet i transakce dokončovací. Pro zachytávání tagů označujících dokončení bus-master transakcí navrhneme jednoduchý monitor. Zapojení jednotlivých prvků verifikačního prostředí je znázorněno na obrázku 4.4.



Obr. 4.4: Schéma verifikačního prostředí pro master variantu endpointu

4.4.9 Bus-master rozhraní a transakce

Vzhledem k tomu, že bus-master rozhraní je vlastně rozhraní interní sběrnice rozšířené o signály `BM_TAG` a `BM_TAG_VLD`, bude použito již hotové rozhraní interní sběrnice a zvlášť nově vytvořené jednoduché rozhraní obsahující jen tyto dva signály. Toto nové rozhraní nemusí obsahovat žádné assertions.

Pro popis transakcí na prvním rozhraní se použije třída popisující běžné transakce interní sběrnice. Pro transakce druhého rozhraní bude vytvořena velmi jednoduchá třída, obsahující jedinou proměnnou `tag`.

4.4.10 Bus-master monitor

Pro zachytávání tagů dokončených transakcí je navržen jednoduchý monitor. Pokud je zapnut, jen sleduje signál `BM_TAG_VLD`, a pokud je tento aktivní, přečte tag ze signálu `BM_TAG` a jako transakci ho předá zaregistrovaným callback funkcím.

4.4.11 Scoreboard pro master verzi

Do scoreboardu nám přibudou dvě callback funkce a tabulka pro bus-master transakce. Jedna z funkcí je zaregistrována v bus-master monitoru a slouží k vyjímání transakcí z této tabulky.

Druhá callback funkce patří nově zapojenému driveru. Čtecí transakce jen ukládá do tabulky transakcí očekávaných na interní sběrnici, protože endpoint by je sem měl beze změny přeposlat. V případě zápisových transakcí však musí nejdříve z uživatelské komponenty vyčíst data k zaspání, a tak funkce tyto transakce převádí na čtecí transakce uživatelského rozhraní a ukládá je do příslušné tabulky. Hlavičku původní zápisové transakce opět ukládá do tabulky transakcí interní sběrnice. Opět je ukládána a později porovnávána jen hlavička, stejně jako u běžných čtecích transakcí.

Callback funkce driveru interní sběrnice má navíc novou funkci – když je jí předána dokončovací transakce, naloží s ní stejně jako se zápisovou, ale navíc její tag uloží do tabulky bus-master transakcí.

4.4.12 Funkční pokrytí pro master verzi

Měření funkčního porkytí se oproti slave verzi liší jen v tom, že u transakcí generovaných na interní sběrnici se všechny podmínky vyžadují i pro dokončovací typ transakcí. Navíc se sledují i transakce generované na bus-master rozhraní, u nichž jsou vyžadovány stejné hodnoty délek a zarovnání jako u transakcí interní sběrnice.

4.5 Řízení testu a předávání parametrů

Nyní když máme navržena verifikační prostředí pro všechny tři komponenty, můžeme se konečně zabývat tím, jak bude verifikace probíhat na nejvyšší úrovni.

Tato verifikační prostředí nám umožňují navrhnout téměř libovolný průběh testu. Generátory, drivery i monitory můžeme kdykoliv zapínat a vypínat, můžeme podrobně nastavovat rozložení jednotlivých typů generovaných transakcí a ovlivňovat všechny jejich parametry, stejně jako rozložení čekacích stavů vkládaných do komunikace drivery a monitory. Tyto parametry můžeme měnit kdykoliv během simulace.

Některé parametry se však za běhu měnit nedají, nebo by to problematické (např. latence paměti u verifikace endpointu), pro změnu některých je dokonce nutné kód znovu zkompilovat (datová šířka). Proto, jak bylo naznačeno v úvodu této kapitoly, bude nutné provést mnoho simulačních běhů s různým nastavením verifikovaných komponent.

Obvykle se všechny parametry testu nastavují jako konstanty v jednom souboru a simulace s nimi pak běží celou dobu. Poté se parametry ručně editují a simulace se spouští znovu. Takto se provede několik simulačních běhů a pokud všechny dopadnou dobře (tzn. není nalezena žádná chyba), může být jednotka prohlášena za úspěšně verifikovanou.

V našem případě je však nutné provést mnoho simulací už jen kvůli různým nastavením verifikovaných komponent, bylo by proto velmi výhodné, kdyby se alespoň ty parametry, u kterých to jde, měnily automaticky v rámci jednoho simulačního běhu.

Předávání parametrů strukturami

Pokud tedy budeme často měnit parametry, je třeba navrhnout efektivní způsob jejich předávání jednotlivým prvkům verifikačního prostředí. Místo množství samostatných proměnných budeme

předávat struktury. Definujeme podobu struktury pro každou skupinu parametrů (např. délky čekacích stavů vkládané driverem nebo rozložení vlastností transakcí). Každá jednotka s parametry nastavitelnými pomocí proměnných pak bude mít tyto proměnné uzavřené v některé z těchto struktur.

Zavedení struktur místo jednotlivých proměnných nemá žádný vliv na funkčnost, ale zdrojové kódy, především kód hlavního modulu testu, budou přehlednější a budou se snáze upravovat.

Průběh testu

Místo souboru s konstantami definujícími parametry testu nyní budeme mít soubor, kde bude pro každou z těchto struktur definováno několik typických skupin hodnot. U struktur definujících délky a pravděpodobnosti vkládání čekacích stavů definujeme například hodnoty tak aby nebyly vkládány žádné čekací stavy, aby byly vkládány jen zřídka a krátké, často ale krátké, nebo naopak velmi dlouhé. U struktur definujících parametry transakcí půjde hlavně o rozložení jejich délek.

V hlavním modulu testu pak budeme během simulace mezi těmito strukturami náhodně vybírat a předávat je příslušným prvkům verifikačního prostředí. Algoritmus programu v hlavním modulu bude vypadat takto:

1. Vytvoř všechny objekty verifikačního prostředí, propoj je a zaregistruj všechny callback funkce.
2. Každému generátoru přiřaď náhodně vybranou sadu parametrů transakcí a každému driveru a monitoru náhodně vybranou definici čekacích stavů.
3. Spusť všechny generátory.
4. Čekej až bude posláno *n* transakcí, kde *n* je náhodné číslo v rozsahu 1 – 200 a poté generátory zastav.
5. S pravděpodobností 1/5 počkej, až všechny transakce projdou systémem a zkontroluj, že jsou tabulky transakcí prázdné.
6. Pokud se v předchozím bodě čekalo, s pravděpodobností 1/3 resetuj verifikovanou komponentu.
7. Pokud je dosaženo 100% pokrytí, nebo se pokrytí za posledních 5000 transakcí nezměnilo, nebo pokud byl dosažen limit počtu transakcí, ukonči simulaci. Jinak běž na bod 2.

Konkrétní hodnoty počtů transakcí a pravděpodobností nejsou tak podstatné, tyto byly zvoleny na základě mých zkušeností tak, aby pokrytí dostatečně rychle konvergovalo k maximální hodnotě a simulace tak netrvala příliš dlouho, ale aby přitom byla komponenta otestována dostatečně pečlivě. Kdyby se během provádění simulací ukázalo, že tyto hodnoty nejsou ideální, lze je samozřejmě snadno upravit.

Při čekání na dokončení transakcí nejde primárně o kontrolu tabulek, ale o zastavování a znovuspouštění toku dat, které může mít na chování verifikované komponenty vliv a proto je nutné ho do testu také zahrnout.

Jako podmínka pro ukončení simulace by v ideálním případě mělo stačit jen dosažení 100% pokrytí. Napsat správně definice všech množin požadovaných hodnot je však obtížné, a to zvláště pro takto parametrizovatelné komponenty, protože SystemVerilog má v konstrukcích pro definice pokrytí jen velmi omezené možnosti vyjadřování podmínek. Proto se při testech spokojíme i s hodnotou pokrytí nedosahující 100 %, pokud bude této hodnotě velmi blízko. Simulaci tedy ukončíme i v případě, že se pokrytí již velmi dlouhou dobu nezvyšuje a vyšší hodnoty tak pravděpodobně nikdy nedosáhne. Podle konkrétní dosažené hodnoty se pak rozhodneme, jestli ji budeme považovat za dostatečnou, nebo je třeba upravit složení vstupů či sledování pokrytí.

5 Výsledky verifikace

Všechna verifikační prostředí a jejich komponenty byly implementovány podle výše uvedeného návrhu a bylo provedeno mnoho simulací s různými konfiguracemi verifikovaných jednotek. Testy probíhaly v simulačním software ModelSim[1] od firmy MentorGraphics.

Seznam všech kombinací nastavitelných parametrů, které byly testovány, je uveden v příloze B. Je vidět, že zvlášť pečlivě byly testovány datové šířky 8 a 64 bitů. Je tomu tak proto, že je v blízké době plánováno nasazení právě těchto šířek v praxi. Parametr endpointu `DATA_REORDER` testován nebyl, protože jeho funkce zatím není implementována. Všechny zde uvedené konfigurace byly verifikovány úspěšně, pokud byla nalezena chyba, byla opravena a poté byla verifikace spuštěna znovu.

Nalezené chyby

Při verifikacích bylo postupně objeveno mnoho chyb, velká část už v průběhu implementace a testování verifikačních prostředí.

Všechny tři jednotky vykazovaly stejné dvě chyby související s protokolem interní sběrnice. První chybou bylo porušení protokolu tím, že signál `DST_RDY_N` byl aktivní i při aktivním resetu. Druhou a vážnější chybou byl předpoklad, že aktivní signál `SOF_N` nebo `EOF_N` automaticky znamená, že je aktivní i `SRC_RDY_N`. Podle specifikace protokolu tomu tak ale nemusí být, právě signál `SRC_RDY_N` určuje platnost ostatních signálů. Pokud tedy není aktivní, mohou mít ostatní signály libovolnou hodnotu. To, že se v jednotce platnost tohoto signálu nekontrolovala, pak v některých případech způsobovalo chybné chování jednotky a generování nesmyslných výstupů.

V komponentách switch a transformer jinak žádná chyba nalezena nebyla. To nebylo velkým překvapením vzhledem k tomu, že jsou tyto jednotky relativně jednoduché, a že již dříve prošly jednoduchými testy.

Avšak endpoint je mnohem složitější a navíc dříve vůbec testován nebyl. Tomu odpovídá i celkový počet 13 nalezených chyb. Některé z nich se projevovaly při téměř jakýchkoli vstupních datech a objevily by je tedy i velmi jednoduché testy. Byly však nalezeny i takové chyby, které se objevily jen za určitých podmínek. Mezi ně patří například chyba, kdy při zápisové transakci a určité posloupnosti signálů `EOF_N`, `SRC_RDY_N` a `WR_RDY`, se aktivoval signál `WR_REQ` o jeden takt dříve, než by měl, a do paměti se tak kromě dat zapsala i poslední část hlavičky, nebo například chyba projevující se zaseknutím endpointu při velké latenci paměti, nízké hodnotě parametru `READ_IN_PROCESS` a příchodu několika čtecích požadavků rychle za sebou.

Význam verifikace

Takovéto chyby se již při jednoduchém testu, ověřujícím chování jen na malém vzorku ručně vytvořených vstupních dat, nemusí vyskytnout, nebo se jejich projevy při mohou ruční kontrole výstupů snadno přehlédnout.

Účelem verifikace však je otestovat jednotku pokud možno všemi možnými kombinacemi vstupů za všech podmínek a navíc výstupy kontroluje automaticky. Proto odhalí i takové chyby, které se projeví jen za velmi specifických a nepravděpodobných podmínek. Při návrhu nějakého složitého hardwarového designu je proto vhodné verifikovat v ideálním případě všechny jeho součásti i design

1 http://www.model.com nebo http://www.mentor.com/products/fv/modelsim/

jako celek. V případě, že je toto až příliš složité, by měly být verifikovány alespoň kritické součásti, jako je právě propojovací systém nebo například řadič DMA přenosů. I malá chyba v takovýchto komponentách totiž může způsobit naprosto nepředvídatelné chování celého systému.

Současný stav

Nyní, po provedené verifikace a odstranění všech nalezených chyb, je pravděpodobné, že již v komponentách propojovacího systému žádné další nejsou. Se stoprocentní jistotou to však tvrdit nelze. Systém by měl splňovat všechny specifikované požadavky, ale nikdy nelze s jistotou vyloučit, že se při jejich sepisování na něco zapomnělo, nebo že byla dokonce udělána chyba v samotném verifikačním prostředí.

Každopádně však bylo díky verifikaci nalezeno mnoho chyb, z nichž na některé by se jinak pravděpodobně přišlo až při nasazení tohoto systému v hardware, a v takovém případě je mnohem obtížnější a časově náročnější zdroj chyby identifikovat.

6 Závěr

Cílem této práce bylo navrhnout, implementovat a provést verifikaci tří komponent generického propojovacího systému. Nejdříve byly popsány obecné principy simulační verifikace a problémy, se kterými se musíme při jejím návrhu vypořádat. Také byly uvedeny základní vlastnosti jazyka System-Verilog, v němž byly verifikační testy implementovány.

V další části práce byl představen systém interních sběrnic, jehož komponenty měly být verifikovány. Tento systém je nově navrženým propojovacím systémem pro hardwarové designy platformy NetCOPE vyvíjené v projektu Liberouter. Jednou z jeho hlavních vlastností je velmi vysoká míra flexibility. Jednotlivé části jeho stromové struktury mohou mít různou datovou šířku a to v rozsahu 8 až 128 bitů, a všechny komponenty, ze kterých se skládá, mají mnoho dalších nastavitelných parametrů, ovlivňujících jejich chování. Velké množství možných konfigurací je sice výhodné pro systém, který je páteří univerzální platformy pro vývoj různých aplikací, při verifikaci to však působilo značné problémy. Přesto však bylo pro každou komponentu navrženo kompletní verifikační prostředí a způsob, jak tyto parametrizovatelné jednotky efektivně testovat.

Hlavním přínosem práce je implementace těchto prostředí a v nich poté provedení samotné verifikace. Ta spočívala ve spuštění mnoha simulací s různým nastavením, při nichž bylo otestováno velké množství konfigurací jednotlivých komponent systému. Během verifikace bylo odhaleno a odstraněno velké množství chyb, z nichž některé by sice jistě byly nalezeny i při běžných testech, některé se ale projevovaly jen za určitých podmínek a pravděpodobně by se na ně přišlo až po delší době skutečného provozu v hardware. Proto lze předpokládat, že vývojový cyklus těchto komponent i celého systému byl díky provedené verifikaci výrazně zkrácen.

V současné době již žádné chyby známy nejsou, což vzhledem k rozsahu provedených testů s velkou pravděpodobností znamená, že se již systém za všech podmínek skutečně chová správně. Protože však šlo o simulační verifikaci, nelze přítomnost dalších chyb nikdy zcela vyloučit. Pokud bychom chtěli korektnost chování systému dokázat, museli bychom provést verifikaci formální, která je však mnohem náročnější.

Během několika týdnů od dokončení této práce bude systém interních sběrnic nasazen do praxe. Kdyby se při ostrém provozu přeci jen objevila nějaká chyba, mohou být verifikační testy snadno upraveny tak, aby přesně simulovaly situaci, při které k chybnému chování dochází, a tak dokázaly chybu reprodukovat v simulátoru, kde se její zdroj mnohem snáze hledá.

Vzhledem k tomu, že prvky verifikačních prostředí byly navrhovány a implementovány s důrazem na jejich rozšiřitelnost a znovupoužitelnost, neměl by být problém použít je beze změny pro sestavení složitějšího prostředí. Do budoucna je totiž plánována i verifikace nejen jednotlivých komponent, ale i celého propojovacího systému spojeného do komplexní komunikační struktury na čipu. Zkušenosti z projektu Liberouter ukazují, že taková verifikace může odhalit další chyby, spojené s běžným provozem verifikovaného systému.

Literatura

[1] KOBIERSKÝ P., et. al. *SystemVerilog verification of VHDL design.* Technical report 35/2007. Praha, CESNET, 2007. Dostupný na URL: <http://www.cesnet.cz/doc/techzpravy/2007/systemverilog-vhdl-verification/>.

[2] SPEAR, C. *SystemVerilog for Verification : A Guide to Learning the Testbench Language Features.* Springer, 2006. ISBN 0-387-27036-1

[3] SUTHERLAND S., DAVIDMANN S., FLAKE P. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Second edition. Springer, 2006. str. 1 – 2. ISBN 0-387-33399-1

[4] *Open Verification Methodology White Paper* [online]. OVM World, 2009. Dostupný na URL: <http://www.ovmworld.org/white_papers.php> [cit. 12. 5. 2009].

[5] *Open Verification Methodology verification IP* [online]. eInfochips, 2009. Dostupný na URL: <http://www.einfochips.com/services/asic/IP/OVM_IP.php> [cit. 12. 5. 2009].

[6] *Incisive Verification IP* [online]. Cadence Design Systems. Dostupný na URL: <http://www.cadence.com/products/fv/verification_ip/pages/default.aspx> [cit. 12. 5. 2009].

[7] MÁLEK, T. *Systém interních sběrnic pro čipy s technologií FPGA.* Diplomová práce. Brno, FIT VUT v Brně, 2008.

[8] *LocalLink Interface Specification*, SP006 (v2.0). Xilinx, 2005. Po registraci dostupný na URL: <http://www.xilinx.com/products/ipcenter/LocalLink_UserInterface.htm>.

Seznam příloh

Příloha A: Seznam požadavků na jednotlivé komponenty propojovacího systému a jejich rozhraní.
Příloha B: Tabulky verifikovaných konfigurací.
Příloha C: CD se všemi zdrojovými kódy a elektronickou verzí tohoto dokumentu.

Příloha A – Požadavky

Zde jsou popsány požadavky (requirements) pro použitá rozhraní a pro jednotlivé verifikované komponenty.

Rozhraní Internal Bus

- Signály DATA, SOF_N a EOF_N jsou platné jen když je aktivní SRC_RDY_N.
- K přenosu dochází, jsou-li zároveň aktivní SRC_RDY_N a DST_RDY_N.
- Mezi EOF_N a SOF_N se nevyskytují žádná data (SRC_RDY_N není aktivní).
- Po každém SOF_N následuje (po určitém čase) signál EOF_N.
- Signál SOF_N nesmí být aktivní po předchozím SOF_N, pokud mezi nimi nebyl aktivní EOF_N.
- Pokud DATA_WIDTH < 128, SOF_N a EOF_N nejsou nikdy aktivní najednou.
- Čtecí transakce nenesou žádná data.
- Zápisové a dokončovací transakce nesou data. Jejich množství odpovídá délce uvedené v hlavičce.
- Transakce nesmí křižovat hranici 4kB stránky.

Rozhraní Endpoint Write

- Signály ADDR, DATA, BE, SOF, EOF jsou platné jen když je aktivní signál REQ.
- K přenosu dochází, jsou-li zároveň aktivní REQ a RDY.
- Signál ADDR obsahuje cílovou adresu transakce. Po každém přenosu je adresa inkrementována o počet bytů datové šířky.
- Signál DATA obsahuje data, která se mají zapsat na adresu danou ADDR.
- Po celou dobu transakce obsahuje signál LENGTH informaci o celkové délce transakce (v bytech).
- V prvním a posledním přenosu transakce jsou podle její adresy a délky nastaveny bity BE (byte enable). V ostatních přenosech mají všechny tyto bity hodnotu 1.
- Na začátku transakce je aktivní signál SOF, na konci EOF.

Rozhraní Endpoint Read

Čtecí požadavky (signály REQ, ARDY_ACCEPT, ADDR, BE, LENGTH, SOF, EOF):

- pokud READ_TYPE = CONTINUAL
  - Signály ADDR, BE, SOF, EOF jsou platné jen když je aktivní signál REQ.
  - K přenosu dochází, jsou-li zároveň aktivní REQ a ARDY_ACCEPT.
  - Signál ADDR obsahuje adresu, ze které se mají číst data. Po každém přenosu je adresa inkrementována o počet bytů datové šířky.
  - V prvním a posledním taktu transakce jsou podle její adresy a délky nastaveny bity BE (byte enable). V ostatních přenosech mají všechny tyto bity hodnotu 1.

    - Na začátku transakce je aktivní signál SOF, na konci EOF.
- pokud READ_TYPE = PACKET
    - Signály ADDR, LENGTH, SOF, EOF jsou platné jen když je aktivní signál REQ.
    - K přenosu dochází, jsou-li zároveň aktivní REQ a ARDY_ACCEPT.
    - Signál ADDR obsahuje adresu, ze které se mají číst data.
    - Signál LENGTH obsahuje počet bytů, které se mají přečíst.
    - Zároveň s REQ jsou aktivní i signály SOF a EOF.

Příchod dat z uživatelské komponenty (signály DATA, SRC_RDY, DST_RDY):

- Signál DATA je platný, jen když je aktivní SRC_RDY.
- K přenosu dochází, jsou-li zároveň aktivní SRC_RDY a DST_RDY.

Transformer

- Jsou splněny požadavky na InternalBus rozhraní (viz výše).
- Každý paket, který přijde na UP port, je přeposlán na DOWN port.
- Každý paket, který přijde na DOWN port, je přeposlán na UP port.
- Pořadí transakcí v každém směru je zachováno.
- Žádný paket se při průchodu transformerem nemění.

Switch

- Jsou splněny požadavky na InternalBus rozhraní (viz výše).
- Master varianta:
    - Příchod paketu na UP port:
        - Pokud cílová adresa paketu spadá do adresového prostoru jednoho z DOWN portů, je na něj paket přeposlán.
        - Jinak je paket zahozen.
    - Příchod paketu na jeden z DOWN portů:
        - Paket globální transakce je automaticky přeposlán na UP port.
        - Pokud cílová adresa paketu spadá mimo adresový prostor switche, je paket přeposlán na UP port.
        - Pokud cílová adresa paketu spadá do adresového prostoru druhého DOWN portu, je na něj přeposlán.
        - Jinak je paket zahozen.
- Slave varianta:
    - Příchod paketu na UP port:
        - Paket je přeposlán na oba DOWN porty.
    - Příchod paketu na jeden z DOWN portů:
        - Paket přeposlán na UP port.
- Žádný paket se při směrování nemění.

Endpoint

Pro typy transakcí jsou zde používány zkratky:

- L2LW (local to local write) – lokální zápisová transakce
- L2LR (local to local read) – lokální čtecí transakce
- L2GW (local to global write) – globální zápisová transakce
- G2LR (global to local read) – globální čtecí transakce
- RDC (read completion) – dokončovací transakce
- RDCL (read completion – last fragment) – dokončovací transakce (poslední fragment odpovědi na čtení)

Požadavky na funkci endpointu jsou uvedeny v následujícím seznamu. V závorkách jsou uváděny hodnoty, které jsou u dané transakce důležité a které musí zůstat po všechny fáze operace stejné.

- Jsou splněny požadavky na jednotlivá rozhraní (viz výše).
- Zápisová operace L2LW:
  - Příchod paketu typu typu L2LW (cílová adresa, délka, data) na rozhraní IB_DOWN.
  - Generování transakce (cílová adresa, délka, data) na rozhraní WRITE.
- Čtecí operace L2LR:
  - Příchod paketu typu L2LR (délka, tag, zdrojová adresa, cílová adresa) na rozhraní IB_DOWN.
  - Generování požadavku (zdrojová adresa, délka) na rozhraní READ.
  - Příchod přečtených dat (data) z uživatelské komponenty na rozhraní READ.
  - Generování paketu RDCL (délka, tag, zdrojová adresa, cílová adresa, data) na rozhraní IB_UP. Pokud DATA_REORDER = true, data jsou přerovnána na cílovou adresu.
- Zápisová Bus Master operace L2GW/L2LW:
  - Příchod paketu L2GW nebo L2LW (zdrojová adresa, cílová adresa, délka, tag) na rozhraní BM.
  - Generování požadavku (zdrojová adresa, délka) na rozhraní READ.
  - Příchod přečtených dat (data) z uživatelské komponenty na rozhraní READ.
  - Generování paketu L2GW resp. L2LW (zdrojová adresa, cílová adresa, délka, data) na rozhraní IB_UP. Pokud DATA_REORDER = true, data jsou přerovnána na cílovou adresu.
  - Generování potvrzení o dokončení transakce (tag) na rozhraní BM.
- Čtecí Bus Master operace G2LR/L2LR:
  - Příchod paketu G2LR nebo L2LR (zdrojová adresa, cílová adresa, délka, tag) na rozhraní BM.
  - Generování stejného paketu na rozhraní IB_UP.
  - Opakuje se 0-N krát:
    - Příchod paketu RDC (zdrojová adresa, cílová adresa, délka, tag, data) na rozhraní IB_DOWN.
    - Generování transakce (cílová adresa, délka, data) na rozhraní WRITE.

    - Příchod paketu RDCL (zdrojová adresa, cílová adresa, délka, tag, data) na rozhraní IB_DOWN.
    - Generování transakce (cílová adresa, délka, data) na rozhraní WRITE.
    - Generování potvrzení o dokončení transakce (tag) na rozhraní BM.
- Pořadí G2LR transakcí musí být dodrženo.
- Pořadí L2GW transakcí musí být dodrženo.
- Pořadí potvrzení o dokončení G2LR transakcí musí být dodrženo.
- Pořadí potvrzení o dokončení L2GW transakcí musí být dodrženo.
- Pokud CONNECTED_TO = SWITCH_SLAVE a cílová adresa transakce přicházející na IB_DOWN rozhraní nespadá do adresového prostoru endpointu, je transakce zahozena.

Příloha B – Výsledky verifikace

V následujících tabulkách jsou uvedeny kombinace parametrů, které byly testovány, a výsledky těchto testů.

Transformer

Parametr	Hodnota						
UP_DATA_WIDTH	64	64	64	32	64	128	128
DOWN_DATA_WIDTH	8	8	8	16	16	8	32
UP_INPUT_BUFFER_ITEMS	0	1	16	30	0	1	19
DOWN_INPUT_BUFFER_ITEMS	0	1	50	0	1	256	111
UP_OUTPUT_PIPE	false	true	true	false	false	true	false
DOWN_OUTPUT_PIPE	false	true	true	true	false	false	true
Výsledek:	OK	OK	OK	OK	OK	OK	OK
Dosažené funkční pokrytí:	100 %	100 %	100 %	100 %	100 %	100 %	100 %

Switch (master)

Parametr	Hodnota							
DATA_WIDTH	64	64	64	8	8	16	32	128
HEADER_NUM	1	5	64	1	33	15	10	1
SWITCH_BASE	0x10000000	0x00000000	0xFFFF0000	0x10000000	0xFFFF0000	0x10000000	0x10000000	0x10000000
SWITCH_LIMIT	0x30000000	0x00020000	0x0000FFFF	0x00020000	0x0000FFFF	0x30000000	0x30000000	0x30000000
DOWN1_BASE	0x10000000	0x00000000	0xFFFF2000	0x00000000	0xFFFF2000	0x10000000	0x10000000	0x10000000
DOWN1_LIMIT	0x10000000	0x00017000	0x00004000	0x00017000	0x00004000	0x10000000	0x10000000	0x10000000
DOWN2_BASE	0x20000000	0x0001B000	0xFFFF8000	0x0001B000	0xFFFF8000	0x20000000	0x20000000	0x20000000
DOWN2_LIMIT	0x20000000	0x00002000	0x00008000	0x00002000	0x00008000	0x20000000	0x20000000	0x20000000
Výsledek:	OK	OK	OK	OK	OK	OK	OK	OK
Funkční pokrytí:	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %

Switch (slave)

Parametr	Hodnota							
DATA_WIDTH	64	64	8	8	8	16	32	128
HEADER_NUM	1	123	1	12	40	1	1	24
Výsledek:	OK	OK	OK	OK	OK	OK	OK	OK
Funkční pokrytí:	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %

Endpoint

128 bitů

Parametr	Hodnota				
BUS_MASTER_ENABLE	false	false	true	true	true
DATA_WIDTH	128	128	128	128	128
ADDR_WIDTH	32	27	32	32	32
ENDPOINT_BASE	0x00001000	0x00001000	0x00001000	0x00001000	0x00000000
ENDPOINT_LIMIT	0x00003000	0x00003000	0x00003000	0x00003000	0x00001FFF
CONNECTED_TO	MASTER	MASTER	MASTER	SLAVE	SLAVE
DATA_REORDER	false	false	false	false	false
READ_TYPE	CONTINUAL	PACKET	PACKET	PACKET	CONTINUAL
READ_IN_PROCESS	1	1	4	1	2
INPUT_BUFFER_SIZE	0	1	0	6	1
OUTPUT_BUFFER_SIZE	0	1	16	10	0
Latence paměti	2	12	64	30	0
Výsledek:	OK	OK	OK	OK	OK
Funkční pokrytí:	100 %	100 %	100 %	100 %	100 %

64 bitů, slave

Parametr	Hodnota					
BUS_MASTER_ENABLE	false	false	false	false	false	false
DATA_WIDTH	64	64	64	64	64	64
ADDR_WIDTH	32	13	32	32	32	32
ENDPOINT_BASE	0x00001000	0x00000000	0x00001000	0x00001000	0x00001000	0x22220000
ENDPOINT_LIMIT	0x00003000	0x00001FFF	0x00003000	0x00003000	0x00003000	0x2223FFFF
CONNECTED_TO	MASTER	SLAVE	MASTER	MASTER	MASTER	SLAVE
DATA_REORDER	false	false	false	false	false	false
READ_TYPE	CONT.	CONT.	PACKET	PACKET	PACKET	PACKET
READ_IN_PROCESS	1	2	1	1	5	3
INPUT_BUFFER_SIZE	0	0	4	0	1	0
OUTPUT_BUFFER_SIZE	0	0	20	0	1	20
Latence paměti	0	0	0	9	30	10
Výsledek:	OK	OK	OK	OK	OK	OK
Funkční pokrytí:	100 %	100 %	100 %	100 %	100 %	99,9 %

64 bitů, master

Parametr	Hodnota					
BUS_MASTER_ENABLE	true	true	true	true	true	true
DATA_WIDTH	64	64	64	64	64	64
ADDR_WIDTH	32	32	32	32	32	32
ENDPOINT_BASE	0x22220000	0x00001000	0x00001000	0x00001000	0x00001000	0x00001000
ENDPOINT_LIMIT	0x2223FFFF	0x00003000	0x00003000	0x00003000	0x00003000	0x00003000
CONNECTED_TO	SLAVE	MASTER	MASTER	MASTER	MASTER	MASTER
DATA_REORDER	false	false	false	false	false	false
READ_TYPE	CONT.	CONT.	CONT.	CONT.	PACKET	PACKET
READ_IN_PROCESS	1	1	8	3	3	5
INPUT_BUFFER_SIZE	0	0	0	1	5	0
OUTPUT_BUFFER_SIZE	0	0	1	10	100	0
Latence paměti	2	0	0	40	0	25
Výsledek:	OK	OK	OK	OK	OK	OK
Funkční pokrytí:	100 %	100 %	99,6 %	99,6 %	100 %	99,9 %

32 bitů

Parametr	Hodnota				
BUS_MASTER_ENABLE	false	false	true	true	true
DATA_WIDTH	32	32	32	32	32
ADDR_WIDTH	16	20	19	16	32
ENDPOINT_BASE	0x00001000	0x00001000	0x00001000	0x00001000	0x00001000
ENDPOINT_LIMIT	0x00003000	0x00003000	0x00003000	0x00003000	0x00003000
CONNECTED_TO	MASTER	MASTER	MASTER	SLAVE	SLAVE
DATA_REORDER	false	false	false	false	false
READ_TYPE	CONTINUAL	PACKET	PACKET	PACKET	CONTINUAL
READ_IN_PROCESS	1	1	5	1	1
INPUT_BUFFER_SIZE	0	1	0	6	1
OUTPUT_BUFFER_SIZE	0	1	10	10	1
Latence paměti	5	5	64	30	2
Výsledek:	OK	OK	OK	OK	OK
Funkční pokrytí:	99,3 %	100 %	99,9 %	100 %	99,3 %

16 bitů

Parametr	Hodnota				
BUS_MASTER_ENABLE	false	false	false	true	true
DATA_WIDTH	16	16	16	16	16
ADDR_WIDTH	32	17	16	13	20
ENDPOINT_BASE	0x4012A500	0x4012A500	0x4012A500	0x4012A500	0x4012A500
ENDPOINT_LIMIT	0x40007FFF	0x40007FFF	0x40007FFF	0x40007FFF	0x40007FFF
CONNECTED_TO	MASTER	MASTER	SLAVE	SLAVE	SLAVE
DATA_REORDER	false	false	false	false	false
READ_TYPE	CONTINUAL	PACKET	CONTINUAL	CONTINUAL	PACKET
READ_IN_PROCESS	1	1	2	1	3
INPUT_BUFFER_SIZE	0	10	1	1	6
OUTPUT_BUFFER_SIZE	0	0	15	1	10
Latence paměti	0	7	4	1	30
Výsledek:	OK	OK	OK	OK	OK
Funkční pokrytí:	98,7 %	100 %	97,8 %	99,1 %	99,9 %

8 bitů

Parametr	Hodnota						
BUS_MASTER_ENABLE	false	false	true	true	true	true	true
DATA_WIDTH	8	8	8	8	8	8	8
ADDR_WIDTH	16	32	20	24	13	13	20
ENDPOINT_BASE	0x4012A500	0x4012A500	0x4012A500	0x4012A500	0x4012A500	0x00000000	0xFFF00000
ENDPOINT_LIMIT	0x40007FFF	0x40007FFF	0x40007FFF	0x40007FFF	0x40007FFF	0x00001FFF	0x000FFFFF
CONNECTED_TO	MASTER	SLAVE	SLAVE	MASTER	MASTER	SLAVE	SLAVE
DATA_REORDER	false	false	false	false	false	false	false
READ_TYPE	CONT.	PACKET	PACKET	CONT.	CONT.	CONT.	CONT.
READ_IN_PROCESS	1	3	3	1	2	1	1
INPUT_BUFFER_SIZE	0	1	0	4	10	16	1
OUTPUT_BUFFER_SIZE	0	5	10	4	1	32	1
Latence paměti	0	5	35	4	5	1	3
Výsledek:	OK	OK	OK	OK	OK	OK	OK
Funkční pokrytí:	98,6 %	99,4 %	99,4 %	99,3 %	99,0 %	99,1 %	98,2 %