VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PAMĚŤOVÝ SUBSYSTÉM V SYSTEMC

SYSTEMC MEMORY SUBSYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. KAMIL MICHL**
AUTHOR

VEDOUCÍ PRÁCE — **Prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Michl Kamil, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **Paměťový subsystém v SystemC**
SystemC Memory Subsystem
Kategorie: Návrh číslicových systémů
Zadání:

1. Seznamte se s C++ knihovnou SystemC, zaměřte se na sekce týkající se TLM (Transaction Level Modeling).
2. Seznamte se s nástrojem Codasip Studio a jazykem CodAL.
3. Navrhněte paměťový manažer pro TLM modely (loosely-timed, approximately-timed), paměťové úložiště a sběrnice podporující protokoly AMBA 3 AHB-Lite, AMBA AXI4-Lite a Codasip Procesor Bus.
4. Navržené komponenty implementujte spolu s jednotkovými testy.
5. Navržené komponenty integrujte jako generátor simulátoru Codasip Studia.
6. Změřte výkonnostní rozdíl původního řešení s novým pomocí simulátoru vybraného procesoru.
7. Zhodnoťte dosažené výsledky a navrhněte možná budoucí rozšíření.

Literatura:

- Black, D. C., Donovan, J., Bunton, B. a Keist, A.: *SystemC: From the Ground Up*. Springer US, 2nd edition, 2010, ISBN: 9780387699578.
- Ghenassia, F.: *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer US, 2005, ISBN 9780387262321.

Při obhajobě semestrální části projektu je požadováno:

- body 1 až 4 zadání (funkční prototyp podporující vybraný protokol bodu 3)

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

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Přikryl Zdeněk, Ing., Ph.D., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 21. října 2019

Abstrakt

Tato práce se zabývá návrhem a implementací paměťového subsystému pro simulaci procesoru. Paměťový subsystém je navržen s pomocí principu modelování na úrovni transakcí. Implementace je provedena v jazyce C++ s využitím knihovny SystemC. Simulace procesoru je převzata ze simulátoru společnosti Codasip. Cílem je vytvoření funkčního propojení procesoru a paměti uvnitř simulátoru. Toto propojení podporuje komunikační protokoly sběrnic AHB3-lite, AXI4-lite, CPB a CPB-lite. Nová implementace tohoto propojení a paměti je zakomponována zpět do původního simulátoru. Výsledný simulátor je otestován pomocí jednotkových testů.

Abstract

This thesis deals with the design and implementation of a processor simulation memory subsystem. The memory subsystem is designed using the Transaction Level Modeling approach. The implementation is done in C++ language utilizing the SystemC library. The processor simulation is adopted from the Codasip company simulator. The objective is to create a functional connection between the processor and the memory inside the simulator. This connection supports communication protocols of AHB3-lite, AXI4-lite, CPB, and CPB-lite buses. The new implementation of the aforementioned connection and the memory is integrated into the original simulator. The resulting simulator is tested using unit tests.

Klíčová slova

SystemC, TLM, abstraktní model procesoru, simulace procesoru, Codasip, Codasip studio, Codasip simulátor, sběrnice, AHB3-lite, AXI4-lite, CPB, CPB-lite

Keywords

SystemC, TLM, abstract processor model, processor simulation, Codasip, Codasip studio, Codasip simulator, bus, AHB3-lite, AXI4-lite, CPB, CPB-lite

Citace

MICHL, Kamil. *Paměťový subsystém v SystemC*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Tomáš Hruška, CSc.

Paměťový subsystém v SystemC

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Prof. Ing. Tomáše Hrušky CSc. Další informace mi poskytl Ing. Zdeněk Přikryl Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Kamil Michl
2. června 2020

Poděkování

Děkuji za spolupráci všem zaměstnancům společnosti Codasip. Zvláště pak Ing. Zdeňku Přikrylovi Ph.D. Dále děkuji za spolupráci Ing. Jiřímu Hynkovi.

Obsah

Kapitola 1

Úvod

Ve stále více a více oblastech se ve světě rozšiřuje použití mikroprocesorů a vestavěných systémů. Vestavěné systémy jsou v dnešní době běžnou praxí a setkáváme se s nimi v životě každým dnem, například v automobilech, domácnostech, kancelářích, obchodech a na mnoha dalších místech. V jádru velkého počtu těchto zařízení se nachází mikroprocesor. Se stále se zvyšujícím počtem použití mikroprocesorů se zvyšuje poptávka po jejich různých typech. Pro každé zařízení je vhodnější jiný typ procesoru. Z tohoto důvodu se velice často používají procesory s aplikačně specifickou instrukční sadou (ASIP, angl. zk. *Application-Specific Instruction set Processor*). Takové procesory jsou náročné na vývoj, ale poskytují vylepšení z pohledu výkonu, spotřeby, velikosti a dalších parametrů.

Vývojem procesorů s aplikačně specifickou instrukční sadou se zabývá například společnost Codasip[11]. Nástroj Codasip Studio slouží k efektivnímu návrhu procesorů. K jejich samotnému popisu je využíván programovací jazyk CodAL. Pomocí nástroje Codasip Studio je možné navrhovat procesory efektivněji, než při použití běžných jazyků a technik pro návrh hardwaru. Codasip Studio také nabízí velkou sadu nástrojů pro práci s vytvořenými návrhy včetně překladačů, testovacích nástrojů, simulátorů a dalších pomocných programů.

Tato práce se zabývá právě simulátorem takto navrženého procesoru, konkrétně jednou z jeho částí. Zaměřuje se na komunikaci mezi samotným procesorem a dalšími zařízeními, které jsou k němu připojeny. Často se k procesoru připojuje jedna nebo více pamětí. Tato zařízení jsou připojena k procesoru pomocí sběrnic. Sběrnice mohou být různých typů, přičemž Codasip Studio podporuje použití čtyř z nich. Jedná se o sběrnice typu AHB3-lite, AXI4-lite CPB-lite a CPB.

Cílem této práce je vylepšit část zmíněného simulátoru od společnosti Codasip. Nová verze simulátoru má využívat speciální knihovnu SystemC pro jazyk C++. Tato knihovna bude využita především pro simulaci komunikace mezi jádrem procesoru a ostatními zařízeními. Knihovna SystemC je celosvětově používaný nástroj pro simulaci a návrh hardwaru. Simulátor by bylo možné jednodušeji upravovat a zakomponovat do komplexnějších simulací. Aktuální implementace rozhraní mezi procesorem a ostatními zařízeními je proprietární, což ztěžuje možnosti uživatelů používat vlastní simulační modely ostatních zařízení.

Kapitola 2 se zabývá obecným pohledem na návrh a simulaci procesorů. Kapitola 3 pojednává o pamětech a sběrnicích, které budou ve výsledném řešení podporovány. V kapitole 4 je podrobněji představena společnost Codasip a její nástroje se zaměřením na simulátor. Kapitola 5 se zaměřuje na knihovnu SystemC a její část pro modelování na úrovni transakcí. Kapitoly 6 a 7 obsahují nový návrh a implementaci upravované části simulátoru s využitím knihovny SystemC. Kapitola 8 se zaměřuje na integraci, testování a rozšiřování nového řešení. Kapitola 9 shrnuje výsledky aktuální práce.

Kapitola 2

Návrh procesorů

Návrh procesorů má velice komplikovaný postup. Existuje více možností jak lze procesor navrhnout a každá z nich je vhodnější pro jiný typ procesoru a jiné vývojáře. Také je ve většině případů nutné vytvořit kromě samotného procesoru dodatečné nástroje, které zpříjemní jeho používání pro uživatele. Dalším nutným krokem bývá testování a verifikace hotového procesoru.

Nejběžnější možností je popsat strukturu a chování procesoru přímo v některém z jazyků pro popis hardwaru [35]. Tento způsob je vhodný například pro menší procesory nebo jednotlivé procesorové komponenty. Vyžaduje také nejméně pomocných nástrojů a pro jeho nasazení je nutný pouze program zajišťující logickou syntézu napsaného kódu. Nevýhodou tohoto způsobu je neúnosně se zvyšující komplexita a nečitelnost výsledného kódu při vytváření složitějších modelů. Další nevýhodou také je, že takto navržený procesor lze jen těžce automatizovaně testovat.

Hlavní nevýhodou tohoto způsobu návrhu je simulace procesoru [29]. Simulace je jedním z nejdůležitějších kroků při návrhu procesoru. Umožňuje totiž hledání chyb, testování, sledování průběhu aplikace spuštěné na procesoru a další pokročilejší funkce. Bez možnosti simulace by bylo potřeba procesor přímo vyrobit, aby bylo možné s ním pracovat. Výroba hardwaru je ale zdlouhavá a nákladná. Částečně se dá tento problém vyřešit použitím FPGA (angl. zk. *Field Programmable Gate Array*) [12]. Z hardwarového popisu procesoru lze vytvořit tzv. bitstream, který po nahrání do FPGA nastaví zařízení tak, aby se chovalo jako zadaný hardware. Toto řešení není tak nákladné, ale stále se zde potýkáme s vysokou časovou náročností. Každá změna hardwaru vyžaduje nový překlad celého projektu a připojení ladícího rozhraní k reálnému hardwaru, což celý postup extrémně zpomalí.

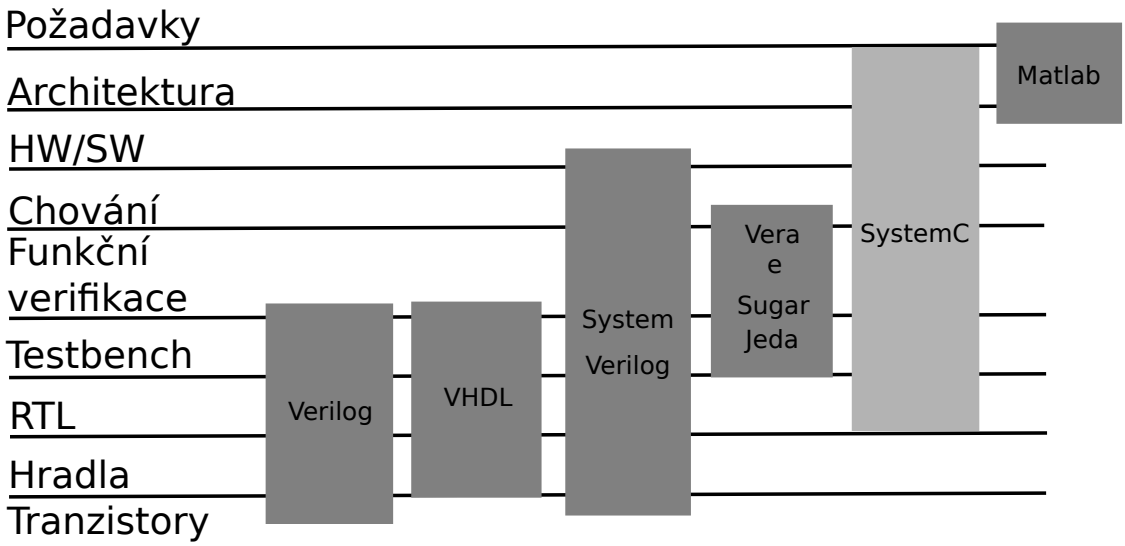
Z tohoto důvodu je vhodná softwarová simulace procesoru. Je výrazně rychlejší, jednodušší, umožňuje větší konfigurovatelnost a přitom stačí na odhalení většiny chyb či nedostatků. Samozřejmě je nutné provádět testy i na reálném hardwaru, aby se zabránilo chybám, které softwarová simulace nedokáže odhalit. Jedná se například o chyby elektronických součástek nebo zpracování nedefinovaných hodnot. Dle návrhu procesoru v jazyce typu HDL (angl. zk. *Hardware Description Language*) [31] by bylo možné simulovat procesor pomocí simulátoru RTL (angl. zk. *Register transfer level*). Tyto simulátory umožňují simulaci libovolných modelů, které jsou popsány v jazyce typu HDL. Nemusí se tedy nutně jednat pouze o procesory. Mezi nejznámější používané simulátory RTL patří například Questa Advanced Simulator, ModelSim, Riviera-PRO, VCS a Xcelium Parallel Logic Simulator [1]. Velkou výhodou takovéto simulace je její podrobnost. Je možné v ní sledovat

[1] [28], [27], [2], [37], [8]

jednotlivé komponenty a signály procesoru v jednotlivých časových okamžicích. Některé nástroje také poskytují různé možnosti verifikace hardwaru. Jedná se například o automatizované spouštění testů, měření pokrytí kódu a podobně. I tato simulace však doplácí na svoji časovou náročnost. Například pro pravidelné testování různých aplikací či funkční verifikaci je samostatně téměř nepoužitelná.

Testování a verifikace je rovněž jedna z nejdůležitějších částí návrhu procesoru. Verifikací rozumíme snahu o dokázání správné funkčnosti dané komponenty. Testování je metoda pro odhalení chyb v komponentě pomocí zkoušení různých vstupních podmínek. Při návrhu hardwaru je často použito funkční verifikace. Podle [43] spočívá funkční verifikace ve vykonávání velkého množství poloautomatizovaných testů. Tyto testy jsou navrženy tak, aby pokryly pokud možno všechny scénáře použití daného hardwaru. Testovací rozhraní je napojeno buď přímo na hardware nebo na simulaci. Dokáže řídit vstupní signály modelu a zároveň kontrolovat jeho výstupní signály. Jednotlivé testovací případy se pak skládají z dvojic vstupů a výstupů, které jsou postupně nastavovány a kontrolovány. Tím je ověřeno, že procesor se chová dle zadané specifikace. V současnosti je pro funkční verifikaci hardwaru nejvíce používaná metodika UVM (angl. zk. *Unified Verification Metodology*) [22] v kombinaci s jazykem SystemVerilog.

Kvůli výše zmíněným nevýhodám se čím dál častěji hledají způsoby, jak návrh procesoru zjednodušit, automatizovat a abstrahovat. V dnešní době existuje řada nástrojů, které se o toto snaží. Jeden z nejpoužívanějších nástrojů je například již zmíněná knihovna SystemC. Mezi další podobné nástroje patří Handel-C nebo Spec-C. Z obrázku 2.1 je patrné, že knihovnu SystemC je možné použít pro popis hardwaru na velkém množství úrovní.



Obrázek 2.1: Srovnání jazyků používaných pro různé úrovně popisu hardwaru. Překresleno z [6].

2.1 Abstraktní model procesoru

Jedna z největších výhod knihovny SystemC a podobných nástrojů je jejich vysoká úroveň abstrakce. Umožňují popisovat hardware pomocí principů známých ze softwarového programování jako jsou například modularita, dědičnost a zapouzdřenost. Z toho plyne velké množství výhod. Především je návrh hardwaru přístupnější pro softwarové vývojáře. Další velkou výhodou je již zmíněná modularita. Ta totiž poskytuje při návrhu procesoru možnost dočasně nahradit hardwarové komponenty softwarovou implementací. Některé komplikovanější části (například aritmeticko-logické jednotky, prediktory skoků a rezervační stanice) je v první fázi návrhu jednoduší napsat softwarově. Softwarové části ovšem nejsou použitelné pro vytvoření hardwaru a musí tedy být v další fázi vývoje nahrazeny hardwarovými komponentami. Důvodem, proč se používají tyto softwarové komponenty, je především možnost dostat model do zkompletované podoby v co možná nejkratším čase. Takto vytvořený model sice není použitelný pro vytvoření hardwaru, ale lze ho úspěšně simulovat, testovat a použít pro návrh aplikací specifických pro daný procesor.

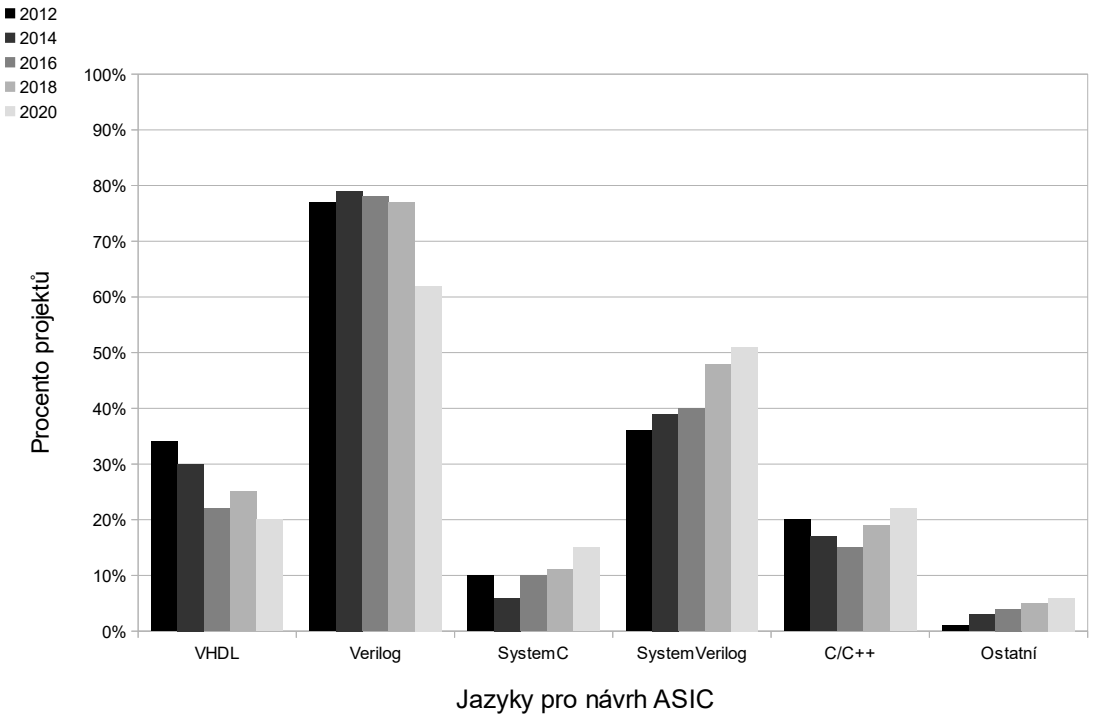
Další výhodou abstraktních modelů procesorů je jejich simulovatelnost. Knihovna SystemC například poskytuje vlastní simulační jádro a prostředí. Není tedy nutné spoléhat na simulátory třetích stran jako je tomu u modelů RTL. Takováto vysokoúrovňová simulace má oproti simulaci RTL řadu výhod i nevýhod. Mezi nevýhody patří především často neexistující grafické rozhraní. Simulace je tedy možná pouze v příkazové řádce, což je pro některé uživatele nepřívětivé prostředí. Další nevýhodou může být vysoká abstrakce simulace, která plyne z abstrakce modelu. Oproti modelu RTL zde nevidíme přímo hardwarové signály a jejich hodnoty, ale pouze jednotlivé moduly s jejich vstupy a výstupy. Vnitřní fungování modulu je často uživateli nepřístupné nebo neodpovídá výsledné hardwarové reprezentaci.

Velkou výhodou této simulace je její rychlost. Právě z důvodu vyšší abstrakce modelu a oproštění se od jednotlivých hardwarových signálů je simulace dostatečně rychlá, aby se prováděla v reálném čase. Tato simulace dokáže ověřit správnost základního konceptu procesoru. Lze jí použít pro odhalení některých chyb v designu. Jelikož ale není dostatečně blízko reálnému hardwaru, mohou se na nižších úrovních abstrakce projevit jisté odlišnosti. Proto je také potřeba simulovat procesor i na úrovni RTL, kde mohou být odhaleny další chyby nebo nedostatky.

Funkční abstraktní model procesoru lze využít i pro účely testování aplikací. Při běžném návrhu procesoru pomocí jazyků typu HDL může softwarová část řešení započít pouze v okamžiku, kdy je model kompletně připraven. Aplikace je možné vyvíjet již při vytvoření abstraktního modelu, který je k dispozici výrazně dříve než hardwarový prototyp. Vývojáři aplikací pro procesor mohou tento model okamžitě využít ke spouštění a ladění svých programů.

Knihovna SystemC je v současnosti nejvíce rozšířený a nejlépe zdokumentovaný nástroj pro tvorbu abstraktních hardwarových modelů. Zároveň přichází s vlastním simulačním jádrem. Z těchto důvodů byla vybrána pro účely této práce. Knihovně SystemC se bude podrobněji věnovat kapitola 5.

Jak lze vidět ze srovnání na obrázku 2.2, knihovna SystemC stále není globálně používána pro přímý popis hardwaru. Nejvíce používanými jazyky jsou predevším Verilog, VHDL a SystemVerilog. Knihovna SystemC je však hojně používána v oblasti abstraktních hardwarových simulací, více než pro samotný hardwarový popis.



Obrázek 2.2: Srovnání jazyků pro návrh procesorů. Překresleno z [15].

2.2 Pomocné nástroje

Knihovna SystemC je výborným nástrojem pro vytvoření abstraktního modelu procesoru. Tento model lze částečně automatizovaně převést na RTL popis hardwaru a následně s ním pracovat na této nízké úrovni abstrakce. Stále je ale nutné k danému procesoru manuálně vytvořit další důležité nástroje. Patří mezi ně především překladač z jazyka symbolických instrukcí do binárního kódu daného procesoru. Dalším důležitým nástrojem je překladač, kde je vstupem některý z vysokoúrovňových jazyků (například C nebo C++). Pomocí těchto překladačů se vytváří programy, které budou na procesoru vykonávány. Jsou tedy jedním z nejdůležitějších procesorových nástrojů. Bez jejich existence by programátor byl nucen psát programy pro daný procesor přímo v binárním kódu. To je velice nepohodlné, časově náročné a pro složitější programy téměř nemožné. Nesmíme zapomenout ani na prostředí pro funkční verifikaci a testování.

Kromě překladačů existuje celá řada podpůrných nástrojů pro práci s procesorem. Nejdůležitější je výše zmíněný simulátor. Dále se jedná o debugger speciálně uzpůsobený cílové architektuře. Dalším nástrojem je tzv. *on-chip debugger* [33], který dokáže komunikovat přímo s hardwarem procesoru pomocí speciálních signálů. Vhodnou pomůckou je také disassembler. Dle [32] je disassembler překladač z binárního kódu aplikace do jazyka symbolických instrukcí. Lze ho využít například při simulaci a ladění bez existence zdrojového kódu aplikace. Užitečná je i řada nástrojů pro práci s binárními soubory [16] pro danou architekturu (například upravená verze programu *objdump* [17]). Nelze zapomenout ani na generátor náhodných programů, verifikační prostředí a další užitečné nástroje.

Ne všechny ze zmíněných pomocných programů musí být pro procesor nutně vytvořeny. Například programy pro práci s binárními soubory nebo generátor náhodných programů je možné z vývoje vynechat. Připravujeme se tím sice o některé možnosti, ale i bez těchto nástrojů je možné procesor bez problémů navrhnout a používat.

Cílem automatizace návrhu procesorů je z abstraktního popisu nejen generovat a simulovat výsledný hardware, ale i automaticky vytvářet různé nástroje. V dnešní době bohužel neexistuje příliš mnoho způsobů, jak zautomatizovat generování procesorových nástrojů. Jedním z nejúspěšnějších pokusů o tuto metodu jsou nástroje od společnosti Codasip, které budou popsány v kapitole 4.

Kapitola 3

Paměti a sběrnice

Procesor není prakticky samostatně použitelná součástka. Jedná se o jádro celého funkčního vestavěného systému. Pro minimální rozumnou funkcionalitu je potřeba připojit k procesoru paměť. K připojení dalších komponent k procesoru jsou využívány sběrnice.

3.1 Paměť

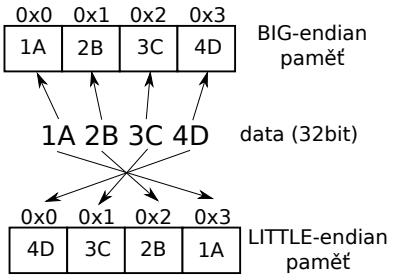
Důležitou vlastností paměti je uchovávání vykonávaného programu a dat. Bez paměti by nebylo možné na procesoru vykonávat žádný program, protože by ho nešlo nikam uložit. Pokud procesor vykonává komplexnější programy, je nutné, aby si uchovával dodatečné informace. Všechna tato data jsou rovněž uložena v paměti.

Paměť většinou podporuje operace čtení dat z konkrétní adresy a zápis dat na určitou adresu. Stejně jako vykonání instrukce na procesoru může trvat několik taktů, paměťové operace také nejsou prováděny okamžitě. Počet taktů, který je potřeba k provedení operace čtení nebo zápisu, se označuje jako latence paměti. Čím menší má paměť latenci, tím rychlejší přístup umožňuje. Nejvhodnější reálné paměti pracují s latencí jednoho taktu. To znamená, že v jednom taktu je zadána operace, a v dalším taktu je připraven její výsledek. Dle [38] dělíme paměťovou strukturu systému na dva typy:

- **Von Neumann architektura paměti** se vyznačuje stejným paměťovým prostorem pro vykonávaný program a ukládaná data. Data a program jsou v takovém případě uloženy na odlišných adresách. Tato architektura umožňuje pohodlnou změnu programu za běhu. Měnit program lze jednoduchým zapsáním programových instrukcí jako dat do správné oblasti paměti.
- **Harvardská architektura paměti** se vyznačuje odděleným prostorem pro data a program. Tato architektura je bezpečnější, protože není možné promíchat spustitelný program s daty. Také její použití je rychlejší, protože lze ve stejném taktu vykonávat operace nad oběma pamětmi zároveň.

Dle [13] má paměť velké množství parametrů. Na fyzické úrovni se jedná především o elektrický způsob provedení. Ten určuje z jakých logických hradel bude paměť vytvořena, zda udržování dat potřebuje konstantní přísun elektřiny a podobně. Pro nás jsou ovšem zajímavé především parametry simulované paměti. Mezi tyto parametry patří především:

- **Velikost paměti**: určuje, jaké množství dat je v ní možné uchovat. Jednotkou velikosti jsou většinou bajty a nebo také LAU (angl. zk. *Least Addressable Unit*).
- **Velikost adresy**: určuje, jak velká je adresa do paměti. Většinou se udává v bitech. Musí být dostatečně velká, aby bylo možné zaadresovat celou velikost paměti.
- **Adresní rozsah**: je velice ovlivněn velikostí. Je ovšem možné používat adresy v jiném rozsahu než od nuly po velikost paměti. Například může být paměť namapována od vyšší adresy než nula. Její velikost poté zůstává stejná, pouze nižší adresy jsou nepřístupné.
- **Velikost slova**: bývá násobkem velikosti jedné LAU či bajtu. Šířka paměťové sběrnice bývá odvozena od tohoto parametru.
- **Endianita**: určuje pořadí bajtů uvnitř jednoho slova. Běžně se používají dva typy endianity, které jsou blíže popsány na obrázku 3.1.
- **Zarovnání**: určuje, na které adresy je možné v paměti přistupovat. Například zarovnání na celé slovo znamená, že adresy, které nejsou násobkem velikosti slova, jsou nepřístupné. Jelikož ale procesor může číst celé slovo, má přístup k celému obsahu paměti, i když ne ke všem adresám.



Obrázek 3.1: Uložení dat v paměti v různých typech endianity.

3.2 Sběrnice

Procesor musí být s pamětí propojen. K připojení dalších zařízení k procesoru slouží sběrnice. Sběrnice je definována sadou signálů a použitým protokolem. Signály jsou v reálném hardwaru reprezentovány fyzickými propoji nebo dráty mezi dvěma zařízeními. Protokol určuje, jak mají jednotlivá zařízení reagovat na určité hodnoty propojovacích signálů.

Dle [14] mají sběrnice řadu parametrů, podle kterých je můžeme zařadit do různých skupin. Sběrnice může být například:

- **Sdílená nebo nesdílená**: u sdílené sběrnice se přenáší všechny typy dat po společné sadě vodičů. U nesdílené sběrnice má každý typ informace samostatný vodič.
- **Synchronní nebo asynchronní**: synchronní sběrnice obsahuje synchronizační signály (typicky například hodinový signál). Podle nich je řízen celý zbytek sběrnice. Asynchronní sběrnice neobsahuje žádné synchronizační signály. Komunikace pak probíhá stylem otázka - odpověď.

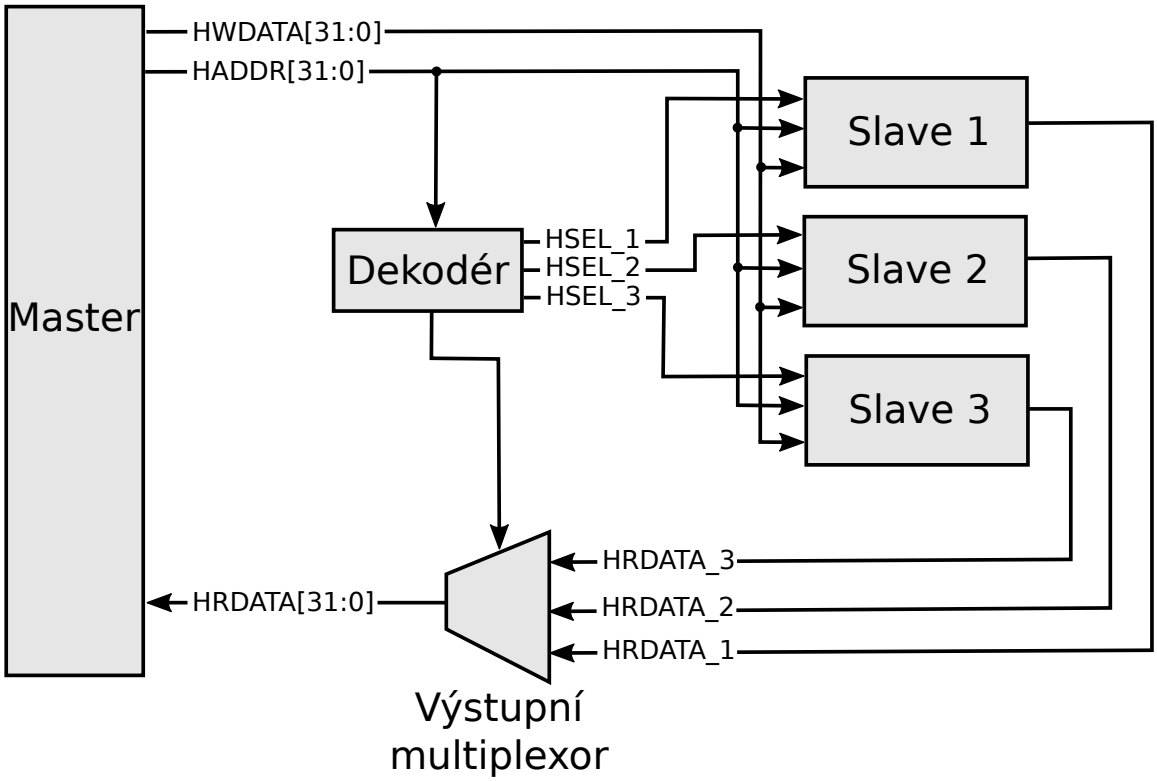
- **Dedikovaná nebo nededikovaná**: dedikovaná sběrnice propojuje právě dvě zařízení. Nededikovaná může propojovat více než dvě zařízení.
- **Paralelní nebo sériová**: sériová sběrnice přenáší zároveň pouze jednu informaci (bit). Paralelní sběrnice může přenášet zároveň více informací (bitů).

Sběrnic existuje velké množství. Každá se hodí pro jiný typ použití. Mezi známé a běžně používané sběrnice patří *USB*, *PCI*, *SATA*, *PCI express* a mnoho dalších [1]. Pro účely této práce budeme uvažovat sběrnice *AHB3-lite*, *AXI4-lite*, *CPB* a *CPB-lite*, které jsou využívány především pro propojení mikroprocesoru s periferními zařízeními, například pamětí. Tyto sběrnice jsou optimalizovány pro co nejvyšší rychlost a výkon.

3.3 Sběrnice AHB3-lite

Sběrnice *AHB3-lite* poskytuje propojení jednoho rozhraní typu master a libovolného počtu rozhraní typu slave. Standard podporuje velikosti dat 32, 64, 128, 256, 512 a 1024 bitů a velikost adresy 32 bitů. Tato sběrnice dokáže přenášet jak jednotlivé transakce, tak bloky transakcí. Je určena pro fungování s velkou hodinovou frekvencí. Informace k této kapitole byly čerpány z [3].

3.3.1 Komponenty a signály

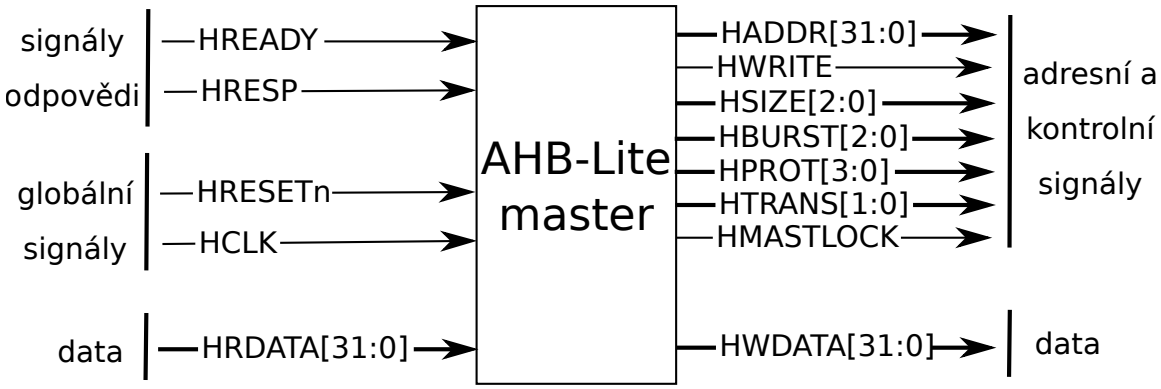


Obrázek 3.2: Schéma propojení komponent sběrnice AHB3-lite. Překresleno z [3].

[1] [39] [30] [34]

Z obrázku 3.2 lze vyčíst, že zde existuje několik klíčových komponent:

- **Master**: rozhraní typu master pro sběrnici inicializuje všechny transakce a očekává odpovědi od rozhraní typu slave.
- **Slave**: rozhraní typu slave pro sběrnici odpovídá na všechny požadavky od rozhraní typu master.
- **Dekodér**: adresový dekodér na základě adresy určuje, kterému zařízení patří daná transakce.
- **Výstupní multiplexor**: multiplexor vybírá výstup z konkrétního zařízení, který se stane vstupem pro rozhraní typu master.



Obrázek 3.3: Schéma signálů sběrnice AHB3-lite z pohledu rozhraní typu master. Překresleno z [3].

Na obrázku 3.3 jsou vyobrazeny jednotlivé signály sběrnice. Tyto signály mají následující sémantiku:

- Signály generované rozhraním typu master:
  - **HADDR[31:0]**: adresa aktuální transakce.
  - **HBURST[2:0]**: typ blokových transakcí.
  - **HMASTLOCK**: indikátor uzamčené transakce.
  - **HPROT[3:0]**: signály pro implementaci dodatečné ochrany a privilegovaného přístupu.
  - **HSIZE[2:0]**: velikost aktuální transakce.
  - **HTRANS[1:0]**: typ aktuální transakce.
  - **HWDATA[n:0]**: data zapisovaná v aktuální transakci (pokud je transakce zápisem). Parametr n určuje velikost datové sběrnice. Může nabývat hodnot 31, 63, 127, 255, 511 a 1023.
  - **HWRITE**: pokud je tento signál v logické jedničce, jedná se o transakci zápisu. V opačném případě se jedná o transakci čtení.

- Signály generované rozhraním typu slave:
  - **HRDATA[n:0]**: obdobně jako signál `HWDATA[n:0]`, pouze tento signál reprezentuje data vyčtené cílovým zařízením.
  - **HREADYOUT**: pokud je v logické jedničce, znamená to, že transakce byla dokončena. V opačném případě je potřeba na dokončení transakce počkat.
  - **HRESP**: slouží jako příznak chyby při vykonávání transakce. Pokud je signál v logické nule, žádná chyba nenastala.
- Globální signály (řízené mimo komponenty sběrnice):
  - **HCLK**: hodinový signál pro časování celé sběrnice.
  - **HRESTn**: reset signál pro celou sběrnici. Jediný signál sběrnice, který je aktivní v logické nule.

3.3.2 Transakce

Každá transakce na této sběrnici má dvě fáze. Adresní fáze určuje typ transakce, všechny potřebné kontrolní signály a adresu pro vykonání operace. Tato fáze musí trvat vždy přesně jeden takt hodin. Ve fázi datové nastává přenos dat po sběrnici. Délku této fáze je možné prodloužit nastavením signálu `HREADYOUT` do logické nuly. Datová fáze musí trvat nejméně jeden takt hodin. Důležitou charakteristikou této sběrnice je, že datová fáze jedné transakce se může překrývat s adresní fází druhé transakce.

Rozhraní typu master musí v každém hodinovém taktu nastavit typ prováděné operace pomocí signálu `HTRANS`, který může nabývat následujících hodnot:

- **IDLE**: jedná se o prázdnou transakci. Rozhraní typu slave musí na tento typ operace odpovědět signálem `HREADYOUT` v logické jedničce a singálem `HRESP` v logické nule. Tato transakce musí být ignorována.
- **BUSY**: tato hodnota reprezentuje vložení čekacího stavu do aktuálního bloku transakcí. Rozhraní typu slave odpovídá ekvivalentně jako při hodnotě `IDLE`. Používá se například, když rozhraní typu master nestíhá zpracovávat přenášená data.
- **NONSEQ**: jedná se o nesekvenční transakci. Značí novou jednotlivou transakci nebo začátek nového bloku transakcí.
- **SEQ**: tato hodnota reprezentuje sekvenční transakci. Značí další transakci v již započatém bloku.

Signál `HBURST` určuje typ bloku, do kterého patří aktuální transakce. Blok transakcí se musí celý vykonat nebo přerušit, než je možné zpracovat další transakce mimo daný blok. Zároveň platí přesná pravidla pro adresy transakcí v jednom bloku. Také v rámci jednoho bloku musí mít všechny transakce shodné hodnoty signálů `HWRITE`, `HSIZE`, `HPROT`, `HMASLOCK` a `HBURST`. Signál `HBURST` může nabývat následujících hodnot:

- **SINGLE**: jedná se o jednotlivou transakci bez jakéhokoliv typu seskupení do bloku. Nelze při ní použít hodnoty `SEQ` a `BUSY` signálu `HTRANS`, jelikož se nejedná o blok.
- **INCR**: tato hodnota reprezentuje blok transakcí o neznámé délce. Dokud hodnota signálu `HTRANS` je `SEQ` nebo `BUSY`, pokračuje se v aktuálním bloku. Adresy jednotlivých transakcí musí pravidelně stoupat od adresy první transakce. Musí se zvyšovat vždy o jednu jednotku velikosti určenou signálem `HSIZE`.

- **INCR4**, **INCR8**, **INCR16**: tyto hodnoty určují blok transakcí pevné délky (4, 8 nebo 16 transakcí). Tento blok musí být po provedení daného počtu transakcí ukončen pomocí hodnoty `IDLE` nebo `NONSEQ` signálu `HTRANS`. Pro adresy jednotlivých instrukcí platí stejná pravidla jako pro blok `INCR`.

- **WRAP4**, **WRAP8**, **WRAP16**: jedná se o bloky s vlastnostmi obdobnými jako `INCR4`, `INCR8` a `INCR16`. Rozdílná jsou pouze pravidla pro inkrementaci adresy mezi jednotlivými transakcemi. Tato pravidla jsou podrobněji popsána ve standardu.

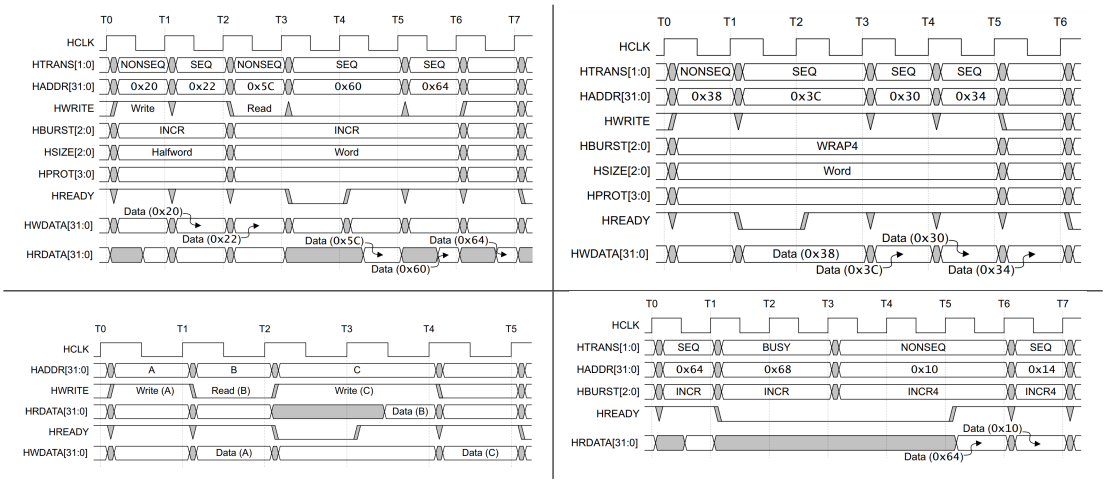
Výsledek transakce v každém taktu závisí na hodnotách signálů `HREADYOUT` a `HRESP`. Mohou nastat tři různé situace:

- **Úspěch**: `HREADYOUT = 1`, `HRESP = 0`. Tato situace značí úspěšné dokončení aktuální transakce.

- **Čekání**: `HREADYOUT = 0`, `HRESP = 0`. V této situaci potřebuje rozhraní typu slave více času na dokončení dané transakce. Pokud tato situace nastane, je nutné, aby do příštího taktu nebyly změněny žádné signály sběrnice generované rozhraním typu master.

- **Chyba**: `HREADYOUT = 0/1`, `HRESP = 1`. Jedná se o chybu na straně rozhraní typu slave. Transakce tedy nemůže být dokončena. Na rozdíl od předchozích situací trvá tento stav dva takty hodin. V první taktu je signál `HREADYOUT` nastaven do logické nuly a ve druhém do logické jedničky. Dva takty trvá tento stav z důvodu dvoufázové logiky sběrnice. V prvním taktu chyby má totiž rozhraní typu master připravenou a odeslanou adresní fázi další transakce. Potřebuje celý jeden takt, aby tuto transakci přerušil. Až ve druhém taktu chyby může poslat adresní fázi další transakce.

Na obrázku 3.4 jsou znázorněny příklady některých ze zmiňovaných typů transakcí. Jedná se o bloky transakcí známé i neznámé délky a vložení čekacích stavů z obou dvou stran komunikace.



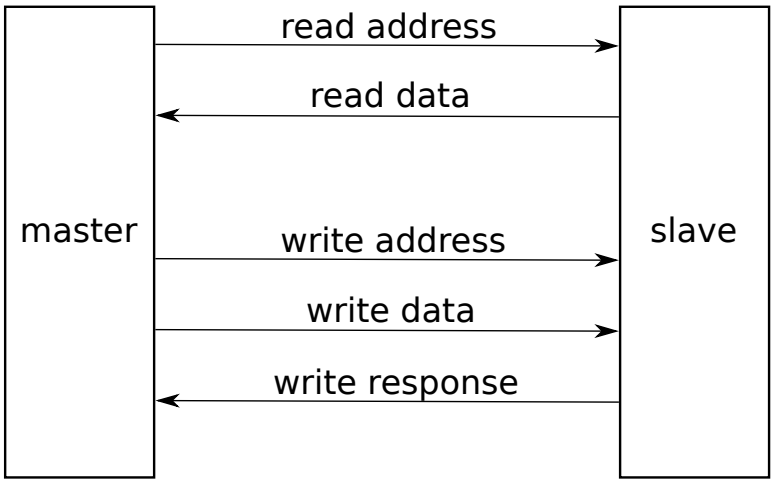
Obrázek 3.4: Příklady transakcí pro AHB3-lite sběrnici [3]

3.4 Sběrnice AXI4-lite

Sběrnice *AXI4-lite* poskytuje, podobně jako sběrnice AHB3-lite, propojení jednoho rozhraní typu master a libovolného počtu rozhraní typu slave. Standard podporuje velikosti dat 32 a 64 bitů a velikost adresy není standardem definována. Tato sběrnice je zjednodušenou verzí sběrnice AXI4 a vyznačuje se především menším počtem kontrolních signálů. Také umožňuje posílání pouze jednotlivých transakcí. Bloky transakcí nejsou podporovány. Oproti sběrnici AHB3-lite se odlišuje především v rozdělení signálů sběrnice do samostatných komunikačních kanálů. Další důležitou vlastností této sběrnice je existence oddělených kanálů pro operace čtení a zápisu. Je tedy možné provádět tyto operace současně, což v některých situacích výrazně zrychluje komunikaci na sběrnici. Informace k této kapitole byly čerpány z [4].

3.4.1 Komunikační kanály

Všechny signály sběrnice, s výjimkou hodinového signálu a signálu reset, jsou rozděleny do pěti komunikačních kanálů, které jsou znázorněny na obrázku 3.5.



Obrázek 3.5: Schéma komunikačních kanálů sběrnice AXI4-lite. Překresleno z [4].

- Kanály přenášející informace směrem od rozhraní typu master k rozhraní typu slave:
  - **write address**: kanál sloužící k posílání adresy a parametrů operace zápisu,
  - **read address**: kanál sloužící k posílání adresy a parametrů operace čtení,
  - **write data**: kanál sloužící k posílání dat operace zápisu.
- Kanály přenášející informace směrem od rozhraní typu slave k rozhraní typu master:
  - **write response**: kanál sloužící k posílání výsledku operace zápisu,
  - **read data**: kanál sloužící k posílání výsledku a dat operace čtení.

Signály každého komunikačního kanálu jsou rozděleny do dvou skupin. Jednu skupinu tvoří signály, které jsou specifické pro daný kanál. Jedná se například o signály adresy pro `write address` a `read address` kanály. Druhou skupinu tvoří tzv. *handshake signály*, které jsou obsaženy k každém komunikačním kanálu. Jejich účelem je zajistit, aby obě komunikující strany mohly odesílat a přijímat informace putující daným komunikačním kanálem. Handshake signály jsou pro každý kanál dva.

Signál `VALID` nastavuje odesilatel a určuje, zda jsou hodnoty ostatních nastavovaných signálů validní. Pokud je signál v logické jedničce, může příjemce zpracovat ostatní signály kanálu a považovat jejich hodnoty za validní. Pokud je signál v logické nule, musí příjemce počkat se zpracováváním informací dokud se hodnota signálu nezmění.

Signál `READY` nastavuje příjemce a určuje, zda je příjemce schopný přijmout informace z komunikačního kanálu. Pokud je signál v logické jedničce, může příjemce zpracovat poslané informace. Pokud je signál v logické nule, odesilatel musí následující hodinový cyklus poslat znovu případné validní informace. Tento postup je opakován dokud se hodnota `READY` signálu nezmění.

Úspěšný přenos dat je tedy možné provést pouze v takovém hodinovém cyklu, ve kterém jsou oba tyto signály nastaveny do logické jedničky. Podle specifikace nesmí být hodnoty těchto signálů na sobě závislé. Například příjemce nesmí nastavit hodnotu signálu `READY` na základě předchozí hodnoty signálu `VALID`. Dalším důležitým pravidlem je nemožnost změnit informace o aktuálně posílané transakci poté, co byla hodnota `VALID` signálu nastavena na logickou jedničku. Pokud v této situaci příjemce není schopen informace zpracovat, odesilatel musí garantovat poslání stejných validních informací v následujícím cyklu.

Zpracování jedné operace vyžaduje posílání informací několika komunikačními kanály. Specifikace zavádí pravidla, jakým způsobem musí být každá operace zpracována a které komunikační kanály mají být využity a v jakém pořadí. Pro operaci čtení je nutné nejprve poslat informace o adrese pomocí kanálu `read address`. V některém z následujících taktů je operace dokončena a výsledek poslán pomocí kanálu `read data`. Pro operaci zápisu je nutné nejprve poslat adresu a data pomocí kanálů `write address` a `write data`. Pořadí aktivace těchto dvou kanálů není důležité a kanály nemusí být aktivovány ve stejném hodinovém cyklu. Operace je zahájena až po přijetí informací z obou kanálů. V některém z následujících taktů je operace dokončena a výsledek předán pomocí kanálu `write response`.

3.4.2 Signály

Signály dělíme na signály jednotlivých komunikačních kanálů a globální signály:

- `write address` kanál:
  - **AWVALID**: `VALID` handshake signál pro tento kanál.
  - **AWREADY**: `READY` handshake signál pro tento kanál.
  - **AWADDR[n:0]**: adresa pro aktuální operaci zápisu. Parametr n určuje velikost adresové sběrnice.
  - **AWPROT[2:0]**: signál pro implementaci dodatečné ochrany a privilegovaného přístupu.

- `write data` kanál:
  - **WVALID**: `VALID` handshake signál pro tento kanál.
  - **WREADY**: `READY` handshake signál pro tento kanál.
  - **WDATA[n:0]**: data zapisovaná aktuální transakcí. Parametr n určuje velikost datové sběrnice.
  - **WSTRB[n:0]**: signál určující, které bajty dat jsou určeny pro zapsání a které mají být ignorovány. Parametr n je určen počtem bajtů datové sběrnice. Například pro datovou sběrnici o velikosti 32 bitů má parametr n hodnotu 4. Každý bit tohoto signálu reprezentuje jeden bajt datové sběrnice. Pokud má bit hodnotu logické jedničky, je reprezentovaný bajt považován za validní a příslušně zpracován příjemcem. Pokud má bit hodnotu logické nuly, musí být reprezentovaný bajt dat příjemcem ignorován.
- `write response` kanál:
  - **BVALID**: `VALID` handshake signál pro tento kanál.
  - **BREADY**: `READY` handshake signál pro tento kanál.
  - **BRESP[1:0]**: signál určující výsledek aktuální operace zápisu. Hodnota nula určuje korektní vykonání operace. Ostatní hodnoty signálu specifikují různé typy chyb.
- `read address` kanál:
  - **ARVALID**: `VALID` handshake signál pro tento kanál.
  - **ARREADY**: `READY` handshake signál pro tento kanál.
  - **ARADDR[n:0]**: adresa pro aktuální operaci čtení. Parametr n určuje velikost adresové sběrnice.
  - **ARPROT[2:0]**: signál pro implementaci dodatečné ochrany a privilegovaného přístupu.
- `read data` kanál:
  - **RVALID**: `VALID` handshake signál pro tento kanál.
  - **RREADY**: `READY` handshake signál pro tento kanál.
  - **RDATA[n:0]**: data přečtená v rámci aktuální operace čtení. Parametr n určuje velikost datové sběrnice.
  - **RRESP[1:0]**: signál určující výsledek aktuální operace. Hodnota nula určuje korektní vykonání operace. Ostatní hodnoty signálu specifikují různé typy chyb.
- globální signály:
  - **ACLK**: hodinový signál pro časování celé sběrnice.
  - **ARESETn**: reset signál pro celou sběrnici. Jediný signál sběrnice, který je aktivní v logické nule.

3.5 Sběrnice CPB a CPB-lite

Codasip Procesor Bus je sběrnice vyvinutá společností Codasip. Podobně jako předchozí sběrnice poskytuje propojení jednoho rozhraní typu master a libovolného počtu rozhraní typu slave. Standard podporuje libovolnou velikost dat v rozmezí od 1 po 1024 bitů a velikost adresy od 1 po 64 bitů. Sběrnice CPB nepodporuje bloky transakcí na rozdíl od sběrnice AHB3-lite ani nerozděluje komunikaci do samostatných kanálů jako sběrnice AXI4-lite. CPB-lite je zjednodušenou verzí sběrnice CPB. Informace k této kapitole byly čerpány z [9].

3.5.1 Signály

Signály sběrnic CPB a CPB-lite jsou následující:

- Signály řízené rozhraním typu master:
  - **AVALID**: signál určující validitu ostatních signálů. Pokud má hodnotu logické jedničky, jedná se o validní transakci. Tento signál plní podobnou funkci, jako signály typu `VALID` ve sběrnici AXI4-lite.
  - **WRITE**: hodnota logické jedničky v tomto signálu indikuje operaci zápisu. Logická nula indikuje operaci čtení.
  - **ADDR[n:0]**: signál obsahující adresu pro aktuální operaci. Parametr n určuje velikost adresové sběrnice.
  - **WDATA[n:0]**: signál obsahující data pro aktuální operaci zápisu. Pokud se jedná o transakci čtení, je tato hodnota ignorována. Parametr n určuje velikost datové sběrnice.
  - **WSTRB[n:0]**: signál určující validitu jednotlivých bajtů zapisovaných dat. Pokud se jedná o transakci čtení, je tato hodnota ignorována. Podrobnější informace o tomto signálu lze najít v kapitole 3.4.2 u stejnojmenného signálu.
- Signály řízené rozhraním typu slave:
  - **AREADY**: hodnota signálu rovná logické jedničce indikuje, že rozhraní může v aktuálním hodinovém cyklu přijmout transakci. V opačném případě musí být validní transakce znovu odeslána v následujícím cyklu rozhraním typu master. Tento signál plní podobnou funkci jako `READY` signály ve sběrnici AXI4-lite.
  - **VALID**: signál indikující dokončení aktuálně prováděné operace. Pokud má signál hodnotu logické jedničky, je operace dokončena a hodnoty signálů udávající výsledek této operace jsou validní.
  - **RESP**: signál určující stav dokončení aktuálně prováděné operace. Hodnota logická nula znamená úspěšné dokončení. Hodnota logická jedna indikuje chybový stav. Tento signál obsahuje validní hodnotu pouze pokud je hodnota signálu `VALID` logická jednička.
  - **RDATA[n:0]**: signál obsahující data získaná v rámci aktuální operace čtení. Parametr n určuje velikost datové sběrnice. Tento signál obsahuje validní hodnotu pouze pokud je hodnota signálu `VALID` logická jednička a pokud aktuálně zpracovávaná operace je typu čtení.

V některých ohledech je tato sběrnice inspirována mechanismy sběrnice AXI4-lite. Například signál `WSTRB` funguje na podobném principu jako stejnojmenný signál sběrnice AXI4-lite. Handshake signály jsou zde rovněž přítomny, ale pouze pro komunikaci směrem od rozhraní typu master k rozhraní typu slave. Opačný směr komunikace, tedy předávání výsledků operací zpět do rozhraní typu master, nemá implementován tento mechanismus. V tomto směru komunikace se používá pouze signál `VALID`, řízený rozhraním typu slave, indikující dokončení operace. Rozhraní typu master musí být schopné přijmout výsledky operace každý hodinový cyklus. V tomto ohledu se sběrnice CPB lišší od sběrnice AXI4-lite, kde může být předání informací pozdrženo pomocí signálů `READY`.

3.5.2 Sběrnice CPB-lite

Sběrnice *CPB-lite* je zjednodušenou verzí sběrnice CPB. Hlavním zjednodušením je odstranění signálů `AREADY` a `VALID` ze specifikace sběrnice. Komunikace na sběrnici se chová stejně jako v případě sběrnice CPB. Hodnoty chybějících signálů jsou automaticky odvozeny. Předpokládá se, že signál `AREADY` je konstantně v logické jedničce. To znamená, že rozhraní typu slave může každý hodinový cyklus přijmout novou transakci. Dále se předpokládá, že rozhraní typu slave dokončí transakci v rámci jednoho hodinového cyklu. To znamená, že rozhraní typu master může přečíst hodnoty signálů `RESP` a `RDATA` vždy následující cyklus po zahájení operace. Při použití tohoto mechanismu není potřeba signál `VALID`, protože rozhraní typu master dokáže předem určit, který hodinový cyklus jsou posílány informace o dokončení operace.

Oproti všem ostatním sběrnicím, zmíněných v této práci, je CPB-lite jedinou sběrnicí, která nepodporuje konfigurovatelnou latenci na rozhraní typu slave. U sběrnic AHB3-lite, AXI4-lite i CPB je možné prodloužit čekání na dokončení operace pomocí specifických signálů. V případě sběrnice CPB-lite neexistuje způsob, jak prodloužit dobu čekání na dokončení transakce. Všechna zařízení typu slave připojená k této sběrnici musí latenci přesně jednoho hodinového cyklu.

Kapitola 4

Návrh procesorů v nástroji Codasip Studio

Codasip je společnost, která se zabývá návrhem procesorů a automatizací tohoto návrhu. Poskytuje nástroje pro co nejjednodušší a intuitivní návrh procesorů. Tyto nástroje jsou zahrnuty v balíčku *Codasip Studio*. Codasip Studio poskytuje možnost napsat zcela nové procesorové jádro nebo upravovat již existující IP (angl. zk. *intelegtual property*) těchto jader. Častou úpravou je například přidání specifického rozšíření instrukční sady. Procesorovým jádrem rozumíme samostatný procesor bez dalších připojených zařízení. Velké množství dodatečných rozšíření či různých typů implementací lze nastavovat přímo v modelu procesoru. Je tedy možné vybrat si z kombinací různých konfiguračních parametrů.

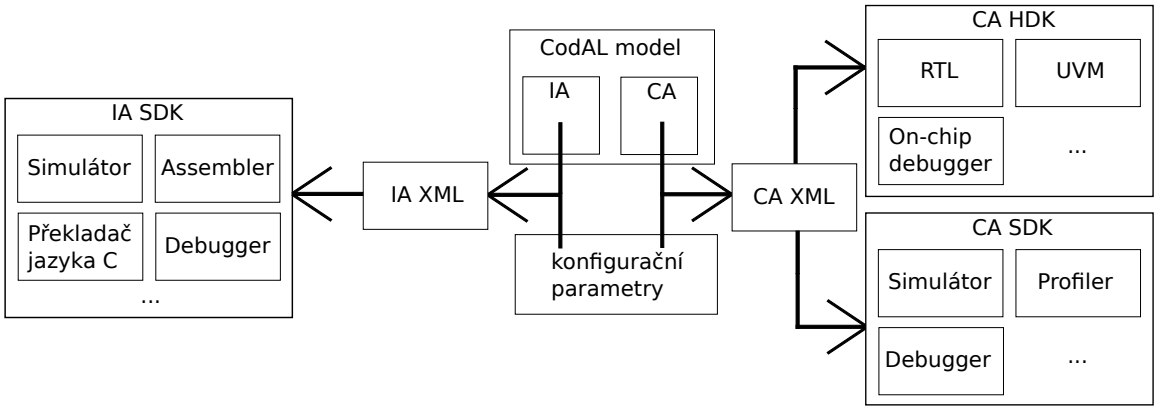
Pro účely této práce bude použito Codasip Studio 8.3.0. Podrobnější informace o společnosti Codasip lze zjistit na internetových stránkách [11]. Dalším zdrojem pro tuto kapitolu je dokumentace pro nástroj Codasip Studio [1].

4.1 CodAL a generované nástroje

CodAL je programovací jazyk, který vytvořil Codasip pro popis procesorů. Jeho výhodou je vysoká úroveň abstrakce. Jazyk CodAL Je založený na standardu ANSI C a poskytuje velké množství nových konstrukcí, které jsou speciálně zaměřeny na popis procesoru. Obsahuje například struktury pro popis instrukční sady, jednotlivých instrukcí, rozhraní, portů, registrů a dalších dodatečných komponent a nastavení. V mnoha ohledech je tento jazyk podobný knihovně SystemC. Slouží pro vytvoření abstraktního modelu procesoru, ale na vyšší úrovni abstrakce než poskytuje knihovna SystemC. Zatímco knihovna SystemC operuje na úrovni modulů, portů a rozhraní, konstrukce jazyka CodAL jsou určeny specificky pro procesory.

Model napsaný v jazyce CodAL je následně přeložen do interní reprezentace ve formě XML (angl.zk. *eXtensible Markup Language*). Z této reprezentace se poté generují různé nástroje umožňující práci s daným procesorovým jádrem. Oproti použití jiných metod je generování pomocných nástrojů procesoru automatické. Všechny potřebné informace lze vyčíst z vytvořeného modelu a případně dodatečné konfigurace. Schéma generování nástrojů z modelu je znázorněno na obrázku 4.1. Tyto generované nástroje se rozdělují do dvou skupin podle toho, z jaké části modelu jsou vytvořeny a k čemu jsou používány.

[1] [10]



Obrázek 4.1: Postup při generování nástrojů v Codasip Studiu.

4.1.1 Instruction Acurate model

Dle [24] slouží *IA* (angl. zk. *Instruction Acurate*) modelování především pro modelování procesoru jakožto systému, který vykonává instrukce. Neuvažuje se v něm žádný typ zpoždění, latence paměti ani časování procesoru samotného. Jeho tvorba je jednoduchá a rychlá. Obsahuje všechny nutné prerekvizity k základnímu simulování procesoru a testování myšlenky modelu. Protože zde chybí informace o časování a zpoždění, je tento typ modelu omezený pouze na vysokoúrovňové ladění. Využití nachází především při vývoji aplikací, kde je softwarovými vývojáři bez problémů používán. Mezi nástroje generované z modelu IA patří:

- **Assembler**: překladač jazyka symbolických instrukcí do binárního kódu dané architektury.
- **Překladač jazyka C:** překlad z jazyka C do binárního kódu dané architektury. Tento překladač je založen na projektu *LLVM* [2], což je soubor znovupoužitelných technologií pro různé typy překladačů, například pro překladač *Clang*.
- **Nástroje pro práci s aplikacemi**: *objdump*, *elfedit*, *strip* a další nástroje z unixových systémů pro práci s binárními aplikacemi. Tyto nástroje však při své práci uvažují danou procesorovou architekturu.
- **Simulátor**: procesorového jádra a případných dalších periferií (například paměti).
- **Debugger**: založený na nástroji *LLDB* [3] (debugger pro LLVM) konfigurovaný pro danou architekturu.
- **Profiler**: nástroj pro získávání statistik a dalších informací z aplikací spuštěných v simulátoru.
- **Co-simulátor**: poskytuje obálku nad klasickým simulátorem. Umožňuje ho používat jako součást větších systémů a simulátorů. V současnosti je podporován co-simulátor pro jazyky C, SystemVerilog a C++ s využitím knihovny SystemC.

[2] [26]
[3] [25]

- **Náhodné programy**: soubor náhodných programů v jazyce symbolických instrukcí sloužící primárně k testování.
- **Další nástroje**: disassembler, testovací scripty, knihovny pro překladač jazyka C a další.

Primárním účelem modelu IA je poskytnout uživateli nástroje umožňující vývoj aplikací pro danou architekturu. Sada těchto nástrojů se také označuje pojmem *SDK* (angl.zk. *Software Developement Kit*). Uživatel si může napsat aplikaci například v jazyce C, spustit ji v simulátoru a ladit podobným způsobem jako program pro běžný počítač.

4.1.2 Cycle Acurate model

Dle [24] slouží *CA* (angl. zk. *Cycle Acurate*) modelování především pro modelování procesoru jakožto hardwarového systému, který obsahuje jednotlivé signály, registry a podobné komponenty. Model CA vzniká většinou rozšířením modelu IA o dodatečnou specifikaci. Tento model by měl obsahovat veškeré informace o časování, latencích a zpožděních. Podobně jako u modelu IA jsou i zde vytvořeny některé z nástrojů patřících do SDK. Jedná se především o simulátor, který je oproti svému protějšku výrazně pomalejší a preciznější, jelikož simuluje i časování. Další obdoby nástrojů z modelu IA jsou zde generovány, pokud dodatečné informace specifikované v modelu CA pomohou k jejich přesnějšímu fungování. Například profiler modelu IA je schopen zjišťovat jaké instrukce a kdy byly provedeny. Profiler modelu CA zjišťuje navíc jakými částmi vykonávání instrukce prošly a zda musel být provoz procesoru pozastaven, například kvůli předcházení hazardů.

Z modelu CA je generována další sada nástrojů, která se označuje jako *HDK* (angl. zk. *Hardware Developement Kit*). Tyto nástroje slouží především k tomu, aby mohl z modelu vzniknout fyzický hardware a bylo možné s ním pohodlně pracovat. Obsahuje především tyto části:

- **RTL**: zdrojové kódy procesoru v některém z jazyků typu HDL. Podporovanými jazyky jsou aktuálně Verilog, SystemVerilog a VHDL.
- **Prostředí pro funkční verifikaci**: založené na UVM, díky kterému lze provést funkční verifikaci výsledného procesoru.
- **On-chip debugger**: nástroj umožňující spuštění a ladění aplikací na reálném hardwaru nebo pomocí simulátoru RTL. Aktuální řešení je založeno na projektu *OpenOCD* [4] a používá speciální hardwarové rozhraní *JTAG* (angl. zk. *Joint Test Action Group* [5]).
- **Scripty pro logickou syntézu**: lze je generovat pro několik podporovaných nástrojů, například od společnosti *Xilinx* [6]. Lze pomocí nich jednoduše provést logickou syntézu modelu RTL do cílové technologie.

[4] [1]
[5] [42]
[6] [41]

4.2 Codasip simulátor

Simulátory vytvořené pomocí Codasip nástrojů jsou dva, jeden pro model IA a druhý pro model CA. Pro každý model je generovaný nový unikátní simulátor v jazyce C++. Simulátor se skládá z části předem připravených komponent a části generovaného kódu. Kromě těchto složek může jádro simulátoru obsahovat propojení s dalšími nástroji, jako je například debugger nebo profiler. Současná implementace simulátoru zde bude více rozvedena, jelikož se jedná o důležitou prerekvizitu pro praktickou část této práce.

První část simulátoru, část komponent (nebo také zdrojů), obsahuje definice jednotlivých částí simulátoru. Tyto definice jsou připravené ve formě hlavičkových souborů C++ a jsou následně použity v druhé části simulátoru. Simulace typu IA i CA (dále jen IA/CA simulace) využívá komponenty ze stejné množiny zdrojů. Mezi komponenty patří běžné hardwarové moduly jako jsou například registry a porty. Velkou část z množiny komponent tvoří moduly sloužící k propojení simulátoru s externí pamětí nebo jinými periferiemi. Tyto komponenty jsou pro tuto práci nejdůležitější a budou detailně rozebrány v následují části práce. Při definici komponent je využito šablonování dostupné v jazyce C++. Parametry šablon jsou použity pro podrobnější specifikaci dané komponenty, například velikost registru nebo latenci paměti. Použití šablon a inline metod dělá výsledný simulátor rychlejším.

Druhá část simulátoru, generovaná část, se skládá ze zdrojových souborů v jazyce C++, které jsou automaticky generovány pro konkrétní procesorový model. Tyto soubory instanciují předem připravené komponenty zmíněné výše a propojují je do funkčních celků. Zároveň je zde generována logika pro interní chování procesoru. Pro simulátor typu IA obsahuje interní logika především dekódování jednotlivých instrukcí na sekvenci sémantických akcí, které jsou procesorem vykonávány (například vyčtení paměti, přepsání programového čítače). Simulátor typu CA obsahuje navíc interní logiku pro propojení procesorových signálů a posílání transakcí po sběrnici. Oba typy simulátorů generují i další pomocné metody, mezi které patří například řízení jednotlivých taktů a resetu. V této generované části jsou přítomny i definice externích periferií. Například paměť se všemi svými rozhraními je zde vygenerována a připojena k procesoru.

4.3 Rozhraní mezi procesorem a pamětí

Nejdůležitější komponenty pro tuto práci slouží ke korektnímu propojení procesoru a paměti. Stejné komponenty lze použít i pro připojení ostatních periferií. Je nutné, aby tyto periferie podporovaly operace čtení a zápisu stejně jako paměť. Zde se ale pro jednoduchost omezíme pouze na klasickou paměť.

Základní propojovací komponentou je *rozhraní* (*interface*). Tato komponenta obsahuje metody pro její propojení s dalšími rozhraními stejného typu. Typ rozhraní je určen primárně velikostí přenášených dat a používaným protokolem. Rozeznáváme dva typy rozhraní, *rozhraní iniciátora* (*initiator interface*) a *rozhraní cíle* (*target interface*). Výhodou těchto rozhraní je, že implementují komunikaci pro režimy IA i CA v jedné třídě. Záleží pouze na tom, jak jsou ve výsledném simulátoru použity.

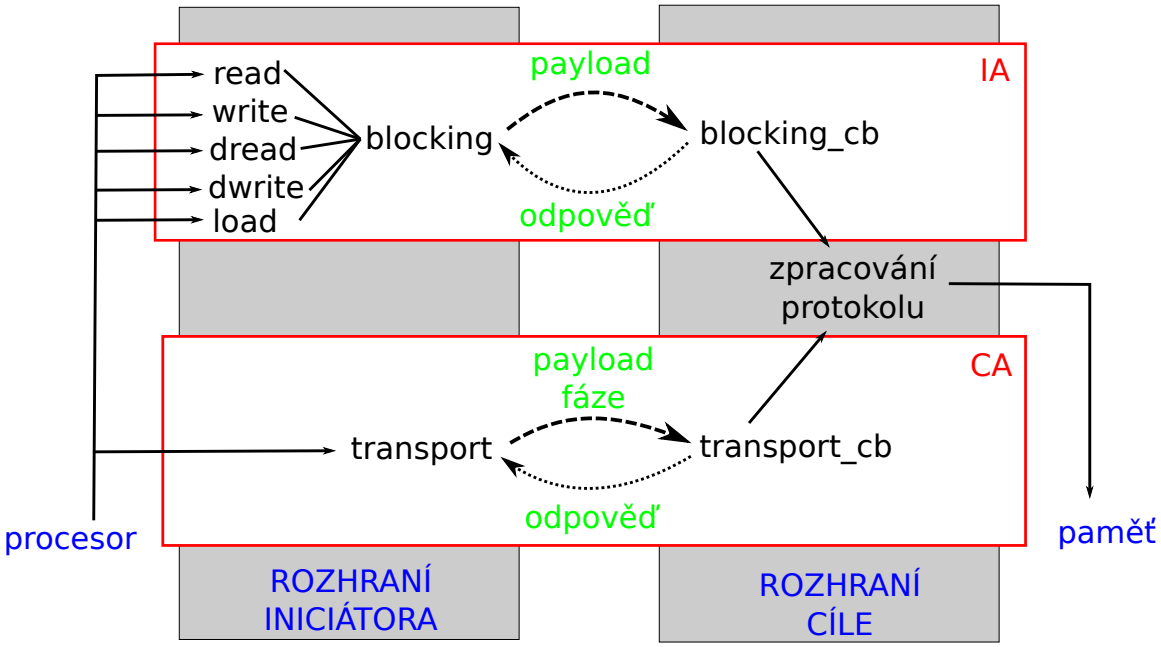
Pro přenos dat mezi jednotlivými rozhraními je použito speciálního datového typu, který je v této kapitole označován jako *Payload*. Obsahuje jednotlivé signály sběrnic a jejich hodnoty pro danou transakci. Stejný objekt je použit pro rozhraní s různými typy sběrnic, liší se pouze jeho vnitřní uspořádání. Payload obsahuje velké množství metod pro zjištění a nastavení hodnot jednotlivých parametrů a signálů. Některé z těchto metod jsou

generické (například přístup k adrese transakce), jiné jsou určeny pro specifickou sběrnici (například přístup ke kontrolním signálům).

Rozhraní obsahují velké množství nastavení a konfigurací. Jedná se například o nastavení zarovnání dat a adresy nebo endianity. Také lze nastavit práva pro přístup do paměti z konkrétního rozhraní:

- **Read only**: povoluje pouze čtení z cílové paměti. Pokusy o zápis končí chybou. Používá se například pro načítání instrukcí.
- **Write only**: povoluje pouze zápis do cílové paměti. Pokusy o čtení končí chybou. Používá se například pro nahrávání aplikace.
- **Read, write**: povoluje čtení i zápis z/do cílové paměti. Používá se například pro instrukce typu `load` a `store`.

Obrázek 4.2 znázorňuje propojení rozhraní iniciátora a cíle. Jednotlivá rozhraní a funkce budou popsány níže.



Obrázek 4.2: Schéma použití rozhraní v Codasip simulátoru.

4.3.1 Rozhraní iniciátora

Rozhraní iniciátora je typicky voláno externě, například z jádra procesoru. Obsahuje metody pro posílání dat v režimu IA. Tyto metody jsou sepsány v následujícím seznamu, kde `T` značí datový typ reprezentující posílaná data a `Resp` značí datový typ odpovědi. Datový typ odpovědi se liší podle použitého rozhraní.

```
// Běžné čtení
T read(const uint64_t address, const size_t count);

// Běžné čtení + předání odpovědi
T read(const uint64_t address, const size_t count, Resp& response);

// Běžný zápis
void write(T data, const uint64_t addr, const size_t count);

// Běžný zápis + předání odpovědi
void write(T data, const uint64_t addr, const size_t count, Resp& response);

// Čtení v debug režimu
T dread(const uint64_t address, const size_t count);

// Zápis v debug režimu
void dwrite(T data, const uint64_t addr, const size_t count);

// Zápis v režimu nahrávání aplikace
void load(T data, const uint64_t addr, const size_t count);
```

Použití těchto metod vede ve výsledku na volání metody `blocking` z definice třídy rozhraní. Tato metoda zajišťuje samotnou komunikaci mezi iniciátorem a cílovým rozhraním v režimu IA.

Metoda `transport` slouží pro komunikaci v režimu CA. Podobně jako v předchozím případě má tato metoda pomocné funkce, které ulehčují její použití. Jelikož v režimu CA musíme zohledňovat architekturu sběrnice a používané signály, jsou tyto pomocné funkce specifické pro konkrétní typ sběrnice. Metoda `transport` obsahuje dodatečný parametr, který určuje fázi sběrnice, pro kterou je daná transakce určena. Obecně mohou mít sběrnice různý počet fází, například adresní a datovou.

4.3.2 Rozhraní cíle

Cílové rozhraní zpracovává požadavky rozhraní iniciátora, ke kterému je připojeno. K tomu primárně slouží tyto metody:

```
void blocking_cb(Payload& p); // Pro režim IA
void transport_cb(Payload& p, const Phase phase); // Pro režim CA
```

Tyto metody jsou volány jako reakce na použití metody `blocking` nebo `transport` v rozhraní iniciátora. Je jim předán objekt typu payload obsahující informace o transakci. Na první pohled nemusí být jasné, že volání obou metod je blokující. Rozdíl mezi nimi je pouze ten, že metoda `blocking_cb` musí zpracovat celou transakci a okamžitě vrátit

výsledek. Oproti tomu metoda `transport_cb` musí dokončit pouze danou fázi transakce. Pro dokončení celé transakce může být nutné provolat tuto metodu vícekrát. Toto chování musí korektně zajistit, ten kdo volá rozhraní iniciátora.

Cílové rozhraní je dále napojeno na instanci paměti. Paměti jsou předány vždy pouze ty informace, které jsou pro ni využitelné. Často se jedná o adresu, data, velikost a příkaz čtení nebo zápisu. Zbytek informací, které jsou specifické pro konkrétní sběrnici, si zpracovává cílové rozhraní interně a paměť je tedy odstíněna od používaného protokolu. Cílové rozhraní se musí vypořádat s libovolnou latencí paměti a také chybami, které mohou při operacích z pamětí nastat. I v těchto speciálních případech musí rozhraní korektně odpovídat podle daného protokolu sběrnice.

4.3.3 Rozhraní v jazyce CodAL

V programovacím jazyce CodAL je rozhraní samostatnou komponentou. Lze definovat rozhraní o různých parametrech. Jednoduché rozhraní lze vytvořit například touto konstrukcí:

```
interface if_test
{
    // šířka adresy, šířka slova, šířka nejmenší adresovatelné jednotky
    bits = { 32, 32, 8 };
    // Použitý protokol a role (MASTER = iniciátor, SLAVE = cíl)
    type = AHB3_LITE:MASTER;
    // Nastavení práv přístupu (R = čtení, W = zápis)
    flag = RW;
    // Nastavení endianity
    endianness = BIG;
};
```

Rozhraní lze propojovat pomocí konstrukce `connect`. V takovém případě je zkontrolováno zda souhlasí potřebné parametry jednotlivých rozhraní. Obě rozhraní musí například používat stejný protokol. Z popisu rozhraní v jazyce CodAL se poté generují příslušné rozhraní iniciátorů a cílů pro procesorový simulátor.

Nevýhodou použití těchto rozhraní pro přenášení dat mezi procesorem a pamětí je proprietální implementace celého paměťového subsystému v simulátoru. Není tedy jednoduše možné napsat vlastní komponentu a připojit ji k procesoru pomocí rozhraní. Řešením je použít již existující a rozšířenou technologii pro implementaci komunikace mezi rozhraními. Pokud by se data přenášela v jednotném formátu, bylo by jednoduší vytvořit vlastní komponentu, která by dokázala korektně komunikovat se zbytkem systému. Popis rozhraní v jazyce CodAL by mohl zůstat nezměněn a upravena by byla pouze implementace rozhraní přímo v simulátoru. Pro řešení tohoto problému lze použít pro implementaci paměťového subsystému v simulátoru knihovna SystemC.

Kapitola 5

SystemC a TLM

SystemC je knihovna pro programovací jazyk *C++* popsaná standardem *IEEE 1666-2011* [21]. Je určena primárně pro možnosti návrhu hardwaru. Zároveň poskytuje simulační jádro, které lze použít pro simulaci modelu vytvořeného pomocí této knihovny. Nejdůležitější části knihovny SystemC knihovny budou v této kapitole podrobněji popsány.

Knihovna SystemC také obsahuje speciální konstrukce pro funkční verifikaci modelů. Další dodatečnou částí knihovny SystemC je implementace standardu *OSCI TLM* (angl. zk. *Open SystemC Initiative Transaction Level Modeling*, dále jen TLM). Tento standard umožňuje jednodušeji modelovat komunikaci mezi jednotlivými částmi systému pomocí speciálních komunikačních kanálů a transakcí. část TLM knihovny SystemC je primárně použita pro modelování komunikace mezi procesorem a pamětí v této práci.

Knihovna SystemC je používána především díky svému unikátnímu postavení ve světě programovacích jazyků a nástrojů. Její velkou výhodou je skutečnost, že je postavená nad jazykem C++ a je tudíž velice blízko vývoji softwaru. Pomocí kombinace knihovny SystemC se základním jazykem C++ je možné navrhovat hardware i software v jednom prostředí a stejném programovacím jazyce. Stejná vlastnost také umožňuje dočasně nahradit některé definice hardwarových komponent softwarovými bloky. Tato skutečnost značně zvyšuje rychlost, s jakou je možno dodat prvotní model výsledného systému. Ten je vhodný například pro účely verifikace a testování. Pokud se jedná o návrh procesoru, je možné z prvotního návrhu již připravovat softwarové aplikace, které na tomto procesoru budou spouštěny. Z časového pohledu je tento postup velmi výhodný, protože testování a případný vývoj softwaru lze začít mnohem dříve, než by bylo možné při klasickém návrhu hardwaru v jazycích typu HDL. Další výhodou je také simulační jádro, které dokáže simulovat vytvořené modely přímo v knihovně SystemC bez nutnosti vysokoúrovňové syntézy. Tato simulace je sice méně přesná, než simulace pomocí simulátorů RTL, ale je oproti ní velice rychlá.

Pro úplné pochopení této kapitoly je potřeba základní znalost jazyka C++. Pro získání podrobnějších informací o knihovně SystemC lze použít další zdroje, ze kterých čerpá i tato práce.

- Kniha SystemC: From the Ground Up [7].
- Kniha Transaction Level Modeling with SystemC [19].
- Kniha The C++ programming language [36].

Postupně se tato kapitola zabývá jednotlivými aspekty knihovny SystemC. Nejprve si zde uvedeme důležité konstrukce, které tato knihovna poskytuje. Dále jsou zde zpracovány základy simulace a použití TLM.

5.1 Konstrukce knihovny SystemC

Knihovna SystemC poskytuje konstrukce pro pohodlnější popis hardwaru. Většina konstrukcí jsou nově definované třídy nebo datové typy v jazyce C++. Tyto konstrukce reprezentují hardwarové komponenty a jejich chování.

5.1.1 Moduly

Základní modelovací jednotkou v knihovně SystemC je *modul*. Každá jednotlivá komponenta modelu, ať už softwarová nebo hardwarová, by měla být modulem. Modul umožňuje uchovávat stav komponenty, popsat její chování a její vnitřní strukturu. Moduly lze hierarchicky skládat do větších komponent a celých systémů.

Z pohledu jazyka C++ je modul objekt, který dědí od speciální třídy `sc_module`. Tato třída zajišťuje základní fungování modulu, například unikátní pojmenování a řízení simulace. Dále poskytuje nástroje pro jednodušší definici chování komponenty, instanciaci vstupních a výstupních rozhraní a další operace.

Definici nového modulu lze provést například tímto způsobem:

```
class nazev_modulu : public sc_core::sc_module
{
    // tělo modulu:
    // proměnné pro uchování stavu
    // metody specifikující chování
    // definice rozhraní
    // deklarace procesů
    // další definice a deklarace

    // Konstruktor
    SC_HAS_PROCESS(nazev_modulu);
    nazev_modulu(sc_core::sc_module_name nazev_instance[,parametry])
        : sc_core::sc_module(nazev_instance)[,inicializace]
        { /* obsah konstruktoru */ }
}
```

Jediným povinným argumentem konstruktoru je jméno instance, které je nutné pro korektní inicializaci třídy `sc_module`. Tělo modulu může obsahovat jak běžné definice a deklarace jazyka C++, tak další speciální konstrukce knihovny SystemC.

5.1.2 Datové typy

Knihovna SystemC poskytuje uživateli k použití nové datové typy. Ve většině případů je doporučeno využívat standardní datové typy jazyka C++, protože práce s nimi je ve většině případů efektivnější. V některých situacích je ale nutné použít datové typy, které lépe vystihují chování hardwaru. Běžné datové typy jazyka C++ jsou z větší části aritmetické. Pro popis hardwaru však často potřebujeme logické a bitové datové typy. Rozdíl je pře-

devším v množství a složitosti prováděných operací a také ve velikosti paměti nutné pro reprezentaci dané hodnoty.

Pokud potřebujeme například součástku čtyřbitového čítače, není vhodné jí definovat jako základní typ jazyka C++. Nejblíže by této implementaci byl datový typ `uint8_t`. Výsledná paměť pro uchování dané hodnoty by byla dvakrát větší než při použití datového typu s přesnou bitovou šířkou. Dalším příkladem by mohla být adresa, jejíž první část slouží pro indexaci zařízení a druhá pro přístup ke konkrétní buňce paměti. Datové typy knihovny SystemC poskytují metody, kterými lze jednoduše tyto dvě části oddělit a pracovat s nimi samostatně. Naproti tomu pro C++ datové typy je nutné si tyto metody definovat ručně s rizikem neoptimální implementace.

Pro práci s logickými hodnotami představuje knihovna SystemC čtyři nové datové typy:

- **sc_bit**: datový typ reprezentující jeden bit s logickými hodnotami nula a jedna. Funguje stejným způsobem jako datový typ `bool` v jazyce C++ a tudíž není často používán.
- **sc_logic**: datový typ reprezentující hodnotu třístavového signálu. Obdobné datové typy můžeme vidět v jazycích typu HDL, například `std_logic` v jazyce VHDL. Může nabývat čtyř různých hodnot:
  - `SC_LOGIC_0`: logická úroveň nula.
  - `SC_LOGIC_1`: logická úroveň jedna.
  - `SC_LOGIC_Z`: stav vysoké impedance.
  - `SC_LOGIC_X`: neznámá hodnota.
- **sc_bv<w>**: datový typ reprezentující vektor bitů. Je odvozen z datového typu `sc_bit` a šířka vektoru je určena hodnotou *w*.
- **sc_lv<w>**: datový typ reprezentující vektor třístavových logických hodnot. Je odvozen z datového typu `sc_logic` a šířka vektoru je opět určena hodnotou *w*.

Oba zmíněné vektorové datové typy podporují další specifické operace. Patří mezi ně například bitové operace `AND`, `OR` a `XOR`. Dále podporují přímý přístup ke konkrétnímu indexu pomocí operátoru hranatých závorek. Také je možné přistoupit ke konkrétnímu úseku indexů pomocí metody `range`. Dále tyto vektorové typy podporují i redukční operace nad svým obsahem.

Další velice užitečné datové typy existují pro vyjádření celých čísel:

- **sc_int<W>**: datový typ pro reprezentaci znaménkového celého čísla. Má libovolný bitový rozsah v rozmezí od 1 do 64 bitů, který je dán hodnotou *w*.
- **sc_uint<W>**: datový typ pro reprezentaci bezznaménkového celého čísla. Má libovolný bitový rozsah v rozmezí od 1 do 64 bitů, který je dán hodnotou *w*.
- **sc_bigint<W>**: datový typ pro reprezentaci znaménkového celého čísla. Má libovolný bitový rozsah větší než 64 bitů, který je dán hodnotou *w*.
- **sc_biguint<W>**: datový typ pro reprezentaci bezznaménkového celého čísla. Má libovolný bitový rozsah větší než 64 bitů, který je dán hodnotou *w*.

Knihovna SystemC definuje i další datové typy, které zde nebudou dále rozvedeny. Za zmínku stojí existence několika datových typů pro čísla ve fixní řádové čárce. Pro vyjádření čísel v plovoucí řádové čárce lze použít typy `float` a `double` jazyka C++.

5.1.3 Porty, rozhraní a kanály

Porty (*ports*), *rozhraní* (*interfaces*) a *kanály* (*channels*) jsou primárním způsobem pro sdílení dat mezi jednotlivými moduly a synchronizaci modulů. V knihovně SystemC existují i speciální konstrukce pro synchronizaci jednotlivých modulů, přes které ale není možné posílat data.

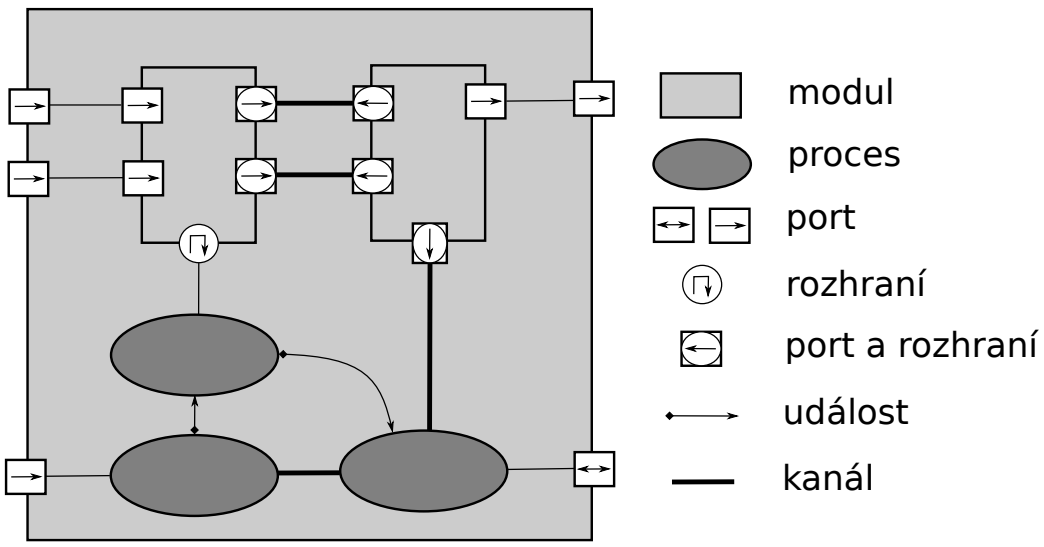
Kanál je speciální typ modulu, který je určen pro komunikaci mezi ostatními moduly. Kanály mají obvykle jeden vstup a jeden výstup. V knihovně SystemC je k dispozici několik standardních kanálů a uživatel si může definovat další. Mezi standardní kanály patří například `sc_mutex` a `sc_semaphore`, které slouží pro synchronizaci mezi moduly. Pro posílání dat slouží například kanály `sc_fifo<T>` a `sc_signal<T>`, kde `T` značí typ přenášených dat. Kanál `sc_signal` simuluje jednoduchý datový tok, který dokáže udržet právě jednu hodnotu. Naproti tomu kanál `sc_fifo` dokáže uchovat více hodnot. To je vhodné například pro bloky transakcí, kdy je potřeba poslat určitý počet dat před samotným vykonáním operace. Kanál `sc_fifo` tedy simuluje frontu typu *FIFO* (angl. zk. *First In First Out*).

Porty jsou konstrukce uvnitř modulů, které umožňují jejich propojení s okolním světem. Každý port musí být propojený se vstupem nebo výstupem komunikačního kanálu. Konkrétně propojuje jedno rozhraní na komunikačním kanálu s daným modulem. S porty se pracuje následovně:

```
sc_port<typ_rozhrani> nazev_portu; // Deklarace portu
instance_modulu.nazev_portu(nazev_kanalu); // Propojení portu s kanálem
```

Rozhraní existují na nižší úrovni komunikace mezi jednotlivými moduly. Jsou použity uvnitř portů a kanálů. Lze pomocí nich definovat vlastní komunikační kanály, které nejsou standardně dostupné v knihovně SystemC. Porty musí být vytvořeny pro specifický typ rozhraní, aby bylo možné je korektně propojit. V knihovně SystemC existují i další způsoby mezimodulové komunikace, které zde nebudou podrobněji popsány, například `sc_export`.

Na obrázku 5.1 lze vidět různé metody propojení jednotlivých komponent knihovny SystemC. Nachází se zde porty, kanály i rozhraní. Také můžeme vidět události a procesy, které jsou součástí simulační části knihovny SystemC.



Obrázek 5.1: Schéma propojení komponentů knihovny SystemC. Překresleno z [6].

5.2 Simulace a procesy

5.2.1 Simulace

Knihovna SystemC poskytuje možnost simulace v ní vytvořených modelů. Hlavní kód programu provádějícího systemC simulaci může vypadat například takto:

```
int sc_main(int argc, char** argv)
{
    // Instanciace a inicializace modulů
    sc_start(); // Začátek simulace
    // Úklid a zobrazení výsledků
    return 0;
}
```

Důležité je, že takto vytvořený program by neměl mít definovanou funkci `main`, jak je v jazyce C++ zvykem. Definice funkce `main` je umístěna uvnitř knihovny SystemC. Pro uživatele je vstupním bodem programu funkce `sc_main`, které jsou předány všechny argumenty programu. Funkce `sc_start` použitá bez parametrů zahájí simulaci bez časového omezení. Program se navrátí z funkce pouze v případě, že již není co simulovat v žádném z vytvořených modulů. Pro omezení doby trvání simulace je možné použít stejnou funkci s dodatečnými parametry. Ty pak určují maximální dobu jejího trvání (například `sc_start(20.0, SC_SEC);`).

SystemC simulace je na rozdíl od reálného hardwarového systému sekvenční a ne paralelní. Paralelismus v knihovně SystemC je pouze simulován pomocí událostí. Simulační jádro plánuje vykonávání jednotlivých procesů a je zodpovědné za posuv simulačního času. Pokud simulace spustí konkrétní proces modulu, tento proces běží nepřerušeně, dokud nepředá řízení zpět simulačnímu jádru. To může nastat buď korektním dokončením procesu nebo pomocí speciální metody `wait`. Kromě přímého posunu simulačního času poskytuje simulace podporu pro tzv. *delta cyklus*. Několik delta cyklů může být provedeno ve stejný simulační čas, jsou však prováděny sekvenčně po sobě. Toto chování je vhodné například pro zpoždění některých komponent při inicializaci, aby nepracovaly s neinicializovanými hodnotami. Další použití je například při sekvenčním propojení modulů. Zde musí být zaručeno, aby všechny vstupy každého modulu byly aktualizovány před vyvoláním výpočtu.

5.2.2 Procesy

Procesy jsou základem definice chování každého modulu. Proces je metoda modulu, která je speciálně zaregistrována v simulačním jádře. Simulační jádro pak tyto metody volá podle průběhu simulace. Procesy musí být registrovány v konstruktoru modulu pomocí speciálních maker. Metoda modulu, která je procesem, nesmí mít žádné parametry a jejím návratovým typem musí být `void`. Existují dva základní typy procesů, `SC_THREAD` a `SC_METHOD`.

`SC_THREAD` je proces, který je spuštěn pouze jednou za simulaci a to většinou na jejím začátku. Tento proces by měl simulovat průběžné chování daného modulu a často tedy obsahuje nekonečnou smyčku. Pokud se provádění této metody dostane do konce, nemůže být proces opakovaně spuštěn. Tento typ procesu může používat metodu `wait` či jiné blokující funkce. Simulační jádro může pozastavený proces rozběhnout na stejném místě, kde se v předchozím běhu vzdal řízení. Pro registraci tohoto typu procesu se používá makro `SC_THREAD(jmeno_metody);`.

`SC_METHOD` je proces, který může být spuštěn kdykoliv během simulace a simulační jádro ho může vyvolat vícekrát. Na rozdíl od `SC_THREAD` tento proces nemůže používat metodu `wait` ani jiné blokující volání. Z pohledu simulačního času je průběh této metody instantní. Neuplyne tedy žádný simulační čas během jejího provádění. Pro registraci tohoto typu procesu se používá makro `SC_METHOD(jmeno_metody);`.

5.2.3 Události

Kdy a za jakých okolností bude předáno řízení konkrétnímu procesu, řeší simulační jádro pomocí *událostí*. Událost může být uživatelem definovaná nebo vestavěná akce, která spustí vykonávání procesu. Mezi vestavěné události patří uplynutí nebo dosažení konkrétního simulačního času, změna hodnoty na komunikačním kanálu a další. Nelze naplánovat dvě stejné události zároveň. Druhé naplánování stejné události přemaže původní plán. Pro specifikaci události slouží datový typ `sc_event`. Nejdůležitější metody této třídy jsou:

- `notify()`: naplánuje událost ve stejném simulačním čase a ve stejném delta cyklu.
- `notify(SC_ZERO_TIME)`: naplánuje událost ve stejném simulačním čase, ale v následujícím delta cyklu.
- `notify(time)`: naplánuje událost ve specifický simulační čas daný parametrem `time`.
- `cancel()`: zruší aktuálně naplánovanou událost.

Procesy jsou citlivé na jednotlivé události. Rozlišujeme citlivost statickou a dynamickou. Dynamická citlivost se dá měnit za běhu simulace. Statická citlivost na události se definuje ihned po registraci procesu v konstruktoru pomocí metody `sensitive`.

U procesu `SC_THREAD` určuje citlivost, na jaké události se má čekat pro znovurozběhnutí procesu. Dynamické citlivosti lze dosáhnout dodatečnými parametry metody `wait`. U procesu `SC_METOD` určuje citlivost, kdy bude tento proces proveden. Pokaždé, když je daná událost naplánována, se spustí proces znovu a celý se vykoná. Pro dynamickou citlivost lze použít metodu `next_trigger`.

5.3 TLM

Modelování na úrovni transakcí (dále jen *TLM*, angl. zk. *Transaction Level Modeling*) je dalším způsobem abstrakce procesorového modelu. Základní knihovna SystemC poskytuje konstrukce pro propojení modulů pomocí datových signálů, podobně jako je tomu na hardwarové úrovni. rozšíření TLM knihovny SystemC poskytuje možnost implementovat tuto komunikaci na abstraktnější úrovni. Při použití složitějších komunikačních prostředků, například sběrnic, je množství propojovacích signálů velké. Bez využití TLM by bylo potřeba každý signál samostatně definovat, ovládat a vyčítat. S použitím TLM je nutné definovat pouze jediný komunikační kanál, přes který lze jednoduše posílat komplexnější zprávy či transakce.

Jak již bylo uvedeno, TLM poskytuje vyšší úroveň abstrakce. Znesnadňuje tím vytvoření výsledného hardwaru, protože není známá přesná hardwarová specifikace použitého komunikačního prostředku. Naproti tomu výrazně ulehčuje vytváření abstraktního modelu. Modelář v první fázi návrhu nemusí znát přesné specifikace použitých sběrnic. Stačí mu pouze základní znalosti propojení a poskytnuté funkcionality. TLM také výrazně zjednodušuje a zrychluje simulaci modelů. Pro simulační jádro je mnohem jednodušší posílat jednu složitější zprávu, než nastavovat řadu samostatných signálů.

Podobně jako nástroje od společnosti Codasip, umožňuje použití TLM dvě různé úrovně modelování:

- **model LT (angl. zk. *lossely timed*)**: ekvivalent modelu IA v nástroji Codasip Studio. Tento model neobsahuje žádné informace a časování. Očekává se, že všechny operace budou vykonány bez zpoždění.
- **model AT (angl. zk. *approximately timed*)**: ekvivalent modelu CA v nástroji Codasip Studio. V tomto modelu nalezneme informace o zpožděních jednotlivých operací a modulů. Očekává se simulované zpoždění některých komponent.

TLM poskytuje několik základních konstrukcí pro dosažení těchto cílů. Především se jedná o struktury typu soket, které slouží jako vstupní a výstupní porty modulů. Dále se jedná o strukturu typu payload, která obsahuje data o konkrétní transakci a je předávána přes struktury typu soket. Moduly pak mohou na tuto transakci různě reagovat, například jejím přeposláním nebo vykonáním specifické akce.

Implementace TLM poskytuje i celou řadu pomocných komponent. Jedná se například o různé implementace struktury typu soket nebo implementace rozšíření pro struktury typu payload. Jednou z nejpoužívanějších pomocných struktur je tzv. *PEQ fronta*, která umožňuje ukládat struktury, nejčastěji typu payload, s časovým razítkem. Tento čas umožňuje přístup k položce fronty až po uplynutí určitého simulačního času. Často se tato komponenta používá pro simulování latence cílových zařízení.

5.3.1 Payload

Objekt typu *payload* (dále jen payload) je určen k přenosu transakce mezi moduly. Z pohledu jazyka C++ se jedná o instanci třídy `tlm_generic_payload`. Tento objekt obsahuje základní informace o dané transakci a metody pro pohodlnou práci s těmito informacemi. Mezi informace obsažené v objektu typu payload patří:

- **Příkaz (m_command)**: identifikuje operaci jako čtení nebo zápis. Obsahuje také speciální hodnotu pro operaci bez efektu.
- **Adresa (m_address)**: určuje cílovou adresu pro zápis nebo čtení dat.
- **Velikost dat (m_length)**: při čtení určuje velikost dat, která mají být přečtena. Při zápisu určuje velikost dat poskytnutých pro zápis.
- **Data (m_data)**: při čtení je zde alokováno místo, kam budou data načtena. Při zápise jsou zde připravena data pro zápis.
- **Odpověď (m_response_status)**: obsahuje stav transakce. Určuje, zda již byla transakce vykonána a jestli byla vykonána korektně. V případě nekorektního provedení může obsahovat i typ chyby.
- **Další informace**: nachází se zde i položky pro přímý přístup do paměti, datovou masku, streaming, apod.

Velkou výhodou třídy `tlm_generic_payload` je její rozšiřitelnost. Ta je dosažena pomocí speciální třídy `tlm_extension`. Rozšíření payloadu se používá pro implementaci specifických protokolů a sběrnic. Typicky obsahuje především kontrolní signály sběrnice a další dodatečné příznaky. Výhodou této metody je, že základní payload je přenositelný mezi

všemi moduly. Pokud modul vyžaduje informace specifické pro danou sběrnici, může si ověřit, zda payload obsahuje dané rozšíření. Pokud ano, může libovolně přistupovat k těmto informacím. Payload je tak možné bez úprav přeposlat přes několik modulů, z nichž některé budou generické a jiné specifické pro konkrétní protokol.

Rozšíření pro payload lze vytvořit a použít například následujícím způsobem:

```
// Vytvoření
class payload_rozsireni : public tlm::tlm_extension<payload_rozsireni>
{
public:
    tlm::tlm_extension_base* clone() const override {}
    void copy_from(tlm::tlm_extension_base const &ext) override {}
    // Proměnné reprezentující data specifického rozšíření
    // Volitelné metody pro přístup k těmto datům
};

// Použití
void Example()
{
    // Vytvoření základního payloadu
    tlm::tlm_generic_payload payload;
    // Přiřazení rozšíření
    payload.set_extension(new payload_rozsireni());
    // Přístup k rozšíření
    payload_rozsireni* rozsireni;
    rozsireni = payload.get_extention<payload_rozsireni>();
    // Kontrola na existenci rozšíření
    if (rozsireni == NULL) return;
    // Odstranění rozšíření
    payload.release_extension<payload_rozsireni>();
}
```

5.3.2 Soket

Payload se vždy přenáší mezi dvěma objekty typu *soket* (dále jen soket). Soket slouží, podobně jako například port nebo rozhraní z knihovny SystemC, pro propojení modulů. Sokety musí být instanciovány v daných modulech a následně propojeny. Poté lze skrz ně posílat informace ve formě objektů typu payload. Podobně jako rozhraní v Codasip simulátoru i zde existují dva různé typy soketů. Jedná se o *soket iniciátora* (*initiator socket*) a *soket cíle* (*target socket*). Pro zjednodušení budeme pracovat s třídami `simple_initiator_socket` a `simple_target_socket`. Tyto třídy výrazně zjednodušují použití soketů. Existují však i nízkoúrovňové implementace těchto tříd. Ty obsahují více možností a konfigurací za cenu zhoršení přehlednosti a čitelnosti.

Soket iniciátora je používán modulem, který zahajuje komunikaci a vytváří požadavky. Podobně jako v Codasip simulátoru i v TLM jsou podporovány dva typy modelování. Existují zde metody použitelné pro LT i AT typy modelů. Nejdůležitějšími metodami jsou:

```
// blocking transport - pro LT modelování
void b_transport(tlm::tlm_generic_payload& p, sc_core::sc_time& delay);

// non-blocking transport - pro AT modelování
sync_enum_type nb_transport_fw(tlm::tlm_generic_payload& p,
    const tlm::tlm_phase& phase, sc_core::sc_time& delay);

// debug transport - pro ladící účely
unsigned transport_dbg(tlm::tlm_generic_payload& p);
```

Metody pro LT a AT modelování se už na první pohled liší. Metoda `b_transport` vyžaduje, aby operace byla kompletně dokončena jednorázově. Metoda `nb_transport_fw` může vykonávat transakci po fázích. Není tedy nutné zpracovat celou transakci v jednom volání. Parametr `phase` určuje fázi vykonávání transakce. Metoda `transport_dbg` funguje podobně jako metoda `b_transport`. Rozdílem je, že metoda `transport_dbg` nesmí mít žádné vedlejší efekty a nepodporuje zpoždění. Její návratovou hodnotou je počet využitých bajtů s datového pole payloadu. Použití nachází například při nahrávání aplikace nebo při přístupu k paměti pomocí debuggeru.

Důležitou součástí použití rozhraní iniciátora je vytvoření korektního objektu typu payload. Před každou operací je potřeba vytvořit payload nový nebo korektně změnit hodnoty starého. Pokud se jedná o specifický protokol, je potřeba vytvořit a nastavit i rozšíření payloadu. Správně nastavený payload je poté možné poslat jednou ze zmíněných metod na rozhraní cíle, kde je dále zpracováván.

Soket cíle je používán modulem, který přijímá a zpracovává transakce. Opět jsou zde podporovány metody pro LT i AT modelování. Nejdůležitějšími metodami jsou:

```
// blocking transport - pro LT modelování
void register_b_transport(MODULE* mod,
    void (MODULE::*cb) (tlm::tlm_generic_payload& p,
        sc_core::sc_time& delay));

// non-blocking transport - pro AT modelování
void register_nb_transport_fw(MODULE* mod,
    sync_enum_type (*MODULE::cb) (tlm::tlm_generic_payload& p,
    const tlm::tlm_phase& phase, sc_core::sc_time& delay));

// debug transport - pro ladící účeli
void register_transport_dbg(MODULE* mod,
    unsigned (MODULE::*cb) (tlm::tlm_generic_payload& p));
```

Těmito registračními funkcemi je možné určit, které metody se zavolají při přijmutí nové transakce. Registrované metody se vykonávají jako reakce na příslušné volání z připojeného rozhraní iniciátora. Úkolem rozhraní cíle je buď transakce zpracovat nebo je přeposlat jinému rozhraní. Povinností rozhraní při zpracování transakce je korektně interpretovat přijatý payload. Dále musí rozhraní provést operaci, kterou payload specifikuje. Nakonec je potřeba vyplnit příslušné informace do payloadu, které mají být odeslány zpět iniciátorovi. Mezi tyto informace patří především kód odpovědi a případně vyčtená data.

Pokud payload obsahuje rozšíření, kterému rozhraní cíle rozumí, je nutné, aby toto rozhraní postupovalo v souladu s daným protokolem. To většinou znamená upravit chování

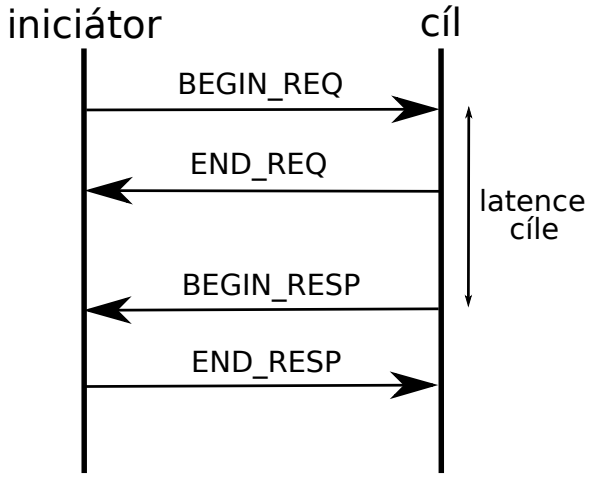
na základě informací získaných z rozšířené sekce payloadu. Pokud tak definuje použitý protokol, je potřeba i nastavovat některé hodnoty v rozšířené sekci.

Kromě payloadu a případně fáze obsahuje volání transportních funkcí i parametr `delay`. Tento parametr by měl být nastavován rozhraním cíle. Jeho hodnota vyjadřuje případné zpoždění, které by reálně vzniklo při vykonávání dané transakce. Rozhraní iniciátora může tento parametr vzít v úvahu a simulovat předané zpoždění.

Kromě zmíněných metod `nb_transport_fw` a `register_nb_transport_fw` existují i metody `nb_transport_bw` a `register_nb_transport_bw`. Tyto metody fungují obdobným způsobem. Jsou umístěny na opačných stranách soketů než první dvojice metod. Slouží pro komunikaci ve směru od cíle k iniciátorovi.

5.3.3 Doporučený protokol

TLM poskytuje doporučený komunikační protokol, který by měl být dodržován oběma komunikujícími stranami při simulaci typu AT. Pokud se jedná o uzavřený systém, není nutné tento protokol dodržovat. Otevřené systémy, které mohou být uživateli rozšiřovány a modifikovány musí podporovat tento protokol nebo definovat protokol vlastní. TLM definuje čtyři základní fáze komunikace: `BEGIN_REQ`, `END_REQ`, `BEGIN_RESP` a `END_RESP`. Doporučený protokol, využívající tyto fáze, je zobrazen na obrázku 5.2.



Obrázek 5.2: Komunikace mezi iniciátorem a cílem podle doporučeného TLM protokolu. Překresleno z [5].

Při simulaci typu AT se muže stát, že modul nedokáže zpracovat některou fázi transakce ihned. I v tomto případě je nutné, aby pro ostatní moduly simulace pokračovala. Modul může použít návratovou hodnotu metod `nb_transport_fw` nebo `nb_transport_bw` pro signalizaci nedokončené operace. Hodnota `TLM_ACCEPTED` označuje transakci přijatou, ale prozatím nezpracovanou. Hodnota `TLM_COMPLETED` označuje dokončení transakce. Pokud transakce nebyla dokončena okamžitě, je strana příjemce, podle doporučeného protokolu, povinna zavolat transportní metodu s příslušnou fází v opačném směru při dokončení transakce. Strana odesilatele může na toto volání příslušně reagovat.

Podle doporučeného protokolu, fáze `BEGIN_REQ` slouží primárně pro poslání adresy a kontrolních informací o transakci. Fáze `END_REQ` je použita pouze pokud cíl nedokáže fázi `BEGIN_REQ` okamžitě zpracovat. Fáze `BEGIN_RESP` je vyvolána na straně cíle a primárně obsahuje výsledek transakce poslané ve fázi `BEGIN_REQ`. Fáze `END_RESP` je použita pouze po-

kud iniciátor nedokáže fázi `BEGIN_RESP` okamžitě zpracovat. Fáze `BEGIN_RESP` a `END_RESP` často slouží i pro předávání dat. Doporučený protokol také specifikuje přesné pořadí použití těchto fází pro konkrétní operaci. Schéma pořadí a směru fází je uvedeno na obrázku 5.2. Dále poskytuje pravidla pro práci s *DMI* (angl. zk. *direct memory access*), endianitou dat, proudovým zpracováním, a podobně.

5.3.4 Definice nových protokolů

Pro modelování a simulaci konkrétní sběrnice nebo komunikačního protokolu často nestačí doporučený TLM protokol. Pro situace, kdy je potřeba předávat více kontrolních signálů, definovat více fází komunikace nebo pozměnit pravidla zpracovávání, umožňuje TLM definovat vlastní komunikační protokoly.

Použitý protokol u každého soketu je možné definovat v jeho šabloně. Tímto způsobem lze odlišit sokety, které komunikují specifickým protokolem. Při propojování soketů iniciátora a cíle je kontrolován použitý protokol na obou stranách komunikace. Lze tedy propojit pouze sokety využívající stejný protokol. Každý protokol je určen strukturou obsahující definice dvou datových typů. Jedná se o datový typ payloadu, který bude mezi sokety přenášen, a datový typ použitý pro specifikaci fáze při simulaci typu AT. Definice nového protokolu může vypadat například následovně:

```
// Definice struktury obsahující nový protokol
// (stejné datové typy jako doporučený protokol)
struct novy_protocol
{
    typedef tlm::tlm_generic_payload tlm_payload_type;
    typedef tlm::tlm_phase tlm_phase_type;
}

// Použití při tvorbě nového soketu
// base určuje modul, ve kterém je soket vytvořen
// konstanta 32 určuje šířku datové sběrnice
tlm_utils::simple_initiator_socket<base, 32, struct novy_protocol>
    initiator_socket;
tlm_utils::simple_target_socket<base, 32, struct novy_protocol>
    target_socket;
```

TLM poskytuje možnost definovat vlastní fáze komunikace pomocí makra `TLM_DECLARE_EXTENDED_PHASE(nazev_fáze);`. Fáze definované tímto makrem lze použít v transportních metodách namísto základních fází. TLM také umožňuje definovat rozšíření základního payloadu (viz kapitola 5.3.1). Díky těmto rozšířením, lze většinu nových protokolů definovat pomocí datových typů doporučeného protokolu (tedy `tlm::tlm_generic_payload` a `tlm::tlm_phase`. Vytváření specifických datových typů pro nový protokol většinou není nutné. Nový protokol není definován pouze výše popsanou strukturou. Nedílnou součástí nově vytvořeného protokolu je specifikace chování obou komunikujících stran při přijetí transakce. Mezi nejdůležitější specifikace chování patří především použití a pořadí definovaných fází. Další důležitou informací je specifikace chování iniciátora a cíle při konkrétních hodnotách jednotlivých položek přijatého payloadu.

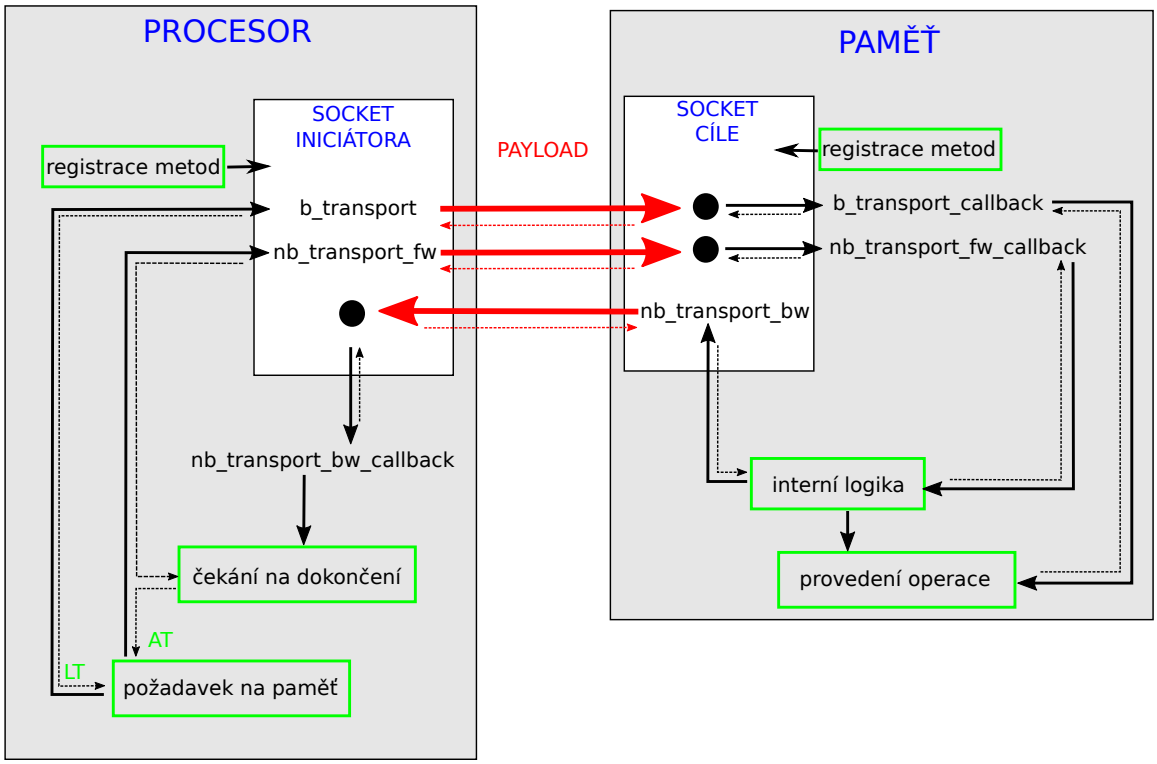
Je výhodné vytvářet nový TLM protokol na základě již existujících specifikací. Vždy při vytváření nového protokolu je potřeba zodpovědět si následující otázky:

- Je možné nový protokol implementovat pomocí existujícího systému payloadu a fází s případnými rozšířeními? Pokud ano, je vhodné použít datové typy a pravidla doporučeného protokolu. Pokud ne, je potřeba nadefinovat vlastní datové typy a celou specifikaci protokolu.
- Postačí pro implementaci protokolu existence základních fází? Pokud ano, je potřeba definovat jakým způsobem se fáze budou využívat, pokud se tento způsob liší od doporučeného protokolu. Pokud ne, je navíc potřeba definovat nové fáze a vytvořit pravidla pro jejich používání.
- Postačí pro implementaci protokolu položky základního payloadu? Pokud ano, je potřeba definovat význam jednotlivých položek, pokud se tento význam liší od doporučeného protokolu. Pokud ne, je potřeba vytvořit rozšíření pro payload a definovat jeho položky. Často je možné namapovat některé informace nového protokolu na položky standardního payloadu, i když jejich původní definice neodpovídá přesně danému použití.

Chování jednotlivých stran komunikace je většinou definováno již existující specifikací modelovaného protokolu. Toto chování tedy nemusí být popsáno vytvořeným TLM protokolem. Je nutné popsat pouze překlad jednotlivých fází a položek payloadu na signály a situace v modelovaném protokolu. Implementace modulů, které používají specifický komunikační protokol, musí být v souladu s definicí tohoto protokolu. Toto základní pravidlo platí pro doporučený TLM protokol i pro nově definované TLM protokoly.

5.3.5 Příklad

V příloze A je uvedena ukázka použití TLM. Skládá se z jednoduchého procesor a jednoduché paměť. Procesor obsahuje rozhraní iniciátora a paměť obsahuje rozhraní cíle. Jedná se implementaci modelu LT, nemáme tedy žádné informace o zpoždění paměti a všechny operace jsou provedeny okamžitě. Procesor se pokusí provést několik operací čtení a zápisu nad pamětí. Paměť provede základní kontrolu a vykoná příslušné operace. Schéma propojení těchto komponent je znázorněno na obrázku 5.3. Pro názornost je zde zakreslená i varianta pro model AT s obousměrnou komunikací. Tento zjednodušený příklad slouží jako počáteční krok k vytvoření funkčního řešení v praktické části práce.



Obrázek 5.3: Schéma komponent a volání metod v praktické ukázce TLM.

Kapitola 6

Návrh řešení

Návrh výsledného řešení byl silně inspirován existující implementací Codasip simulátoru. Rozhraní instanciovaných komponent bylo z velké části zachováno z důvodu jednodušší integrace nového paměťového subsystému do simulátoru. Interní fungování všech komponent bylo kompletně přepracováno. Při návrhu bylo využito znalosti komponent z knihovny SystemC.

Pro správnou funkčnost paměťového subsystému musí být pro simulátor připraveny tři základní komponenty:

- rozhraní iniciátora (pro každou sběrnici),
- rozhraní cíle (pro každou sběrnici),
- paměť.

Tyto komponenty mají některé společné vlastnosti. Jedná se především o velikost používané adresy, velikost nejmenší adresovatelné jednotky, velikost slova a endianitu. Tyto společné vlastnosti by měly být odděleny od implementace konkrétních komponent. Podobně jsou řešeny i základní vlastnosti komponent vzhledem k simulačnímu jádru. Jedná se například o jméno a jednoznačnou identifikaci komponenty. Při použití knihovny SystemC je vhodné, aby každá nově vytvořená komponenta simulátoru byla zároveň i SystemC modulem. Oba zmíněné typy rozhraní mají některé společné vlastnosti. Jedná se například o nastavení zarovnání adresy a zarovnání dat. Tyto společné vlastnosti obou typů rozhraní je možné abstrahovat.

Zmíněné komponenty jsou základem funkčního paměťového subsystému. Kromě nich je potřeba navrhnout celou řadu pomocných komponent a konstrukcí. Jednou z nejdůležitějších pomocných komponent je správce paměti pro použité payloady. Cílem této komponenty je zrychlení běhu simulace pomocí vylepšené správy paměti použitých payloadů. Mezi další pomocné funkcionality patří výběr vhodného datového typu na základě velikosti dat a jednotný přístup k různým typům zdrojů simulátoru. Pomocné součásti budou podrobněji rozebrány v kapitole 7.

6.1 Návrh rozhraní iniciátora

Každé rozhraní iniciátora musí obsahovat mechanismus pro propojení a výměnu dat s rozhraním cíle. Část TLM knihovny SystemC poskytuje nástroje, kterými lze tohoto docílit. Jedná se o třídu `simple_initiator_socket` (viz kapitola 5.3.2). Ta poskytuje možnost připojení cílového rozhraní pomocí metody `bind`. K připojenému rozhraní lze přistupovat a volat jeho metody pro přenos dat. Rozhraní pro konkrétní sběrnice a jejich protokoly lze vytvořit rozšířením takto obecného rozhraní.

Rozhraní pro model IA je z větší části totožné pro všechny použité protokoly. Existuje zde ale zásadní odlišnost, která komplikuje jednotné použití. Jedná se o odpověď na transakci, která má různou sémantiku pro jednotlivé protokoly. Tento problém lze vyřešit použitím speciálních transformačních metod, které jsou závislé na použitém protokolu. Ostatní funkcionalita pro IA simulaci je totožná pro všechny použité protokoly a může být abstrahována. Metody tohoto rozhraní, které jsou přístupné ze simulátoru, je vhodné zachovat. Seznam těchto metod se nachází v kapitole 4.3.1. Zachováním veřejně přístupných metod se lze vyhnout rozsáhlým úpravám při integraci do simulátoru. Při testování je možné použít některé z existujících testů, které jsou implementovány v Codasip nástrojích.

Rozhraní pro model CA musí být specifické pro konkrétní protokol, jelikož mezi protokoly existuje velké množství rozdílů. V CA simulaci se setkáváme s jednotlivými signály sběrnic. Transportní metody rozhraní musí tyto signály reflektovat. Podobně jako u modelu IA by i zde měly být zachovány veřejně přístupné metody. Některé části rozhraní lze navrhnout jednotně, nezávisle na použitých protokolech. Především se jedná o práci s TLM sokety. Vytváření a zpracování payloadů v CA simulaci je specifické pro protokol. Jedinou výjimkou je kopírování dat z a do payloadu. Data transakcí všech podporovaných protokolů jsou v Codasip simulátoru reprezentovány celočíselným datovým typem o dostatečné bitové šířce. V payloadu jsou data reprezentována ve formátu bajtového pole. Je tedy možné použít jednotný způsob pro převod mezi těmito reprezentacemi bez ohledu na použitý protokol.

6.2 Návrh komunikace mezi rozhraními

V původním řešení paměťového subsystému byl pro přenos dat mezi rozhraními použit speciální datový typ `Payload`. Knihovna SystemC poskytuje vlastní způsob přenosu dat pomocí třídy `tlm_generic_payload` (viz kapitola 5.3.1). Vytváření nových objektů typu payload pro jednotlivé operace může výrazně zpomalit chod simulátoru. Tento problém řeší tzv. *manažer payloadů* (*payload manager*), který umožňuje používat vytvořené payloady opakovaně.

Pro účely IA simulace je dostačující základní payload poskytnutý knihovnou SystemC. Obsahuje všechny potřebné položky pro vykonání operace v paměti. Mezi tyto informace patří především typ operace, adresa, data a jejich velikost. Protože v IA simulaci nezohledňujeme vlastnosti použitého protokolu, nejsou zde přítomny žádné specifické signály sběrnice. Jedinou výjimkou je signál indikující chybový stav operace. Tento signál lze abstrahovat pomocí položky odpovědi v TLM payloadu.

Pro účely CA simulace je základní payload nedostačující. Knihovna SystemC poskytuje možnost rozšíření základního payloadu. Primárně by mělo každé rozšíření obsahovat signály sběrnice, které nelze reprezentovat položkami základního payloadu. Dále může obsahovat pomocné metody pro ulehčení práce s obsaženými informacemi. Některé signály sběrnice mohou být reprezentovány položkami základního payloadu. Pro tyto signály musí existovat

konverzní funkce. Podobný mechanismus lze použít v IA simulaci pro konverzi odpovědi do správného formátu.

Výhodou použití třídy `tlm_generic_payload` je především možnost přenášet stejnou strukturu dat mezi rozhraními bez ohledu na použitý protokol nebo rozšíření. Mezi procesor a paměť lze připojit další komponenty, například vyrovnávací paměť nebo zařízení pro arbitraci operací. Některé z komponent nemusí vždy znát použitý protokol, přesto mohou s transakcemi pracovat na obecné úrovni.

6.3 Návrh rozhraní cíle a paměti

Rozhraní cíle přijímá informace od rozhraní iniciátora pomocí payloadu. Nově obdržený payload musí být okamžitě zpracován. Je možné provádět v payloadu i změny, například vyplnění přečtených dat nebo nastavení příznaku chyby. Pokud se jedná o CA simulaci, rozhraní cíle je zodpovědné za zpracování informací specifických pro použitý komunikační protokol. Provádí se také kontroly transakcí, například kontrola korektnosti payloadu nebo kontrola oprávnění pro provedení dané operace.

Rozhraní cíle by mělo interně obsahovat také vlastní rozhraní iniciátora. Tímto způsobem usnadníme práci uživateli, který simulátor používá zároveň s debuggerem. Debugger díky tomuto principu může číst informace přímo z cílového rozhraní. Lze tak například ověřit, zda nejsou chyby v komunikaci mezi rozhraními. Vestavěné rozhraní iniciátora je povinno poskytovat pouze přístupové metody pro IA simulaci. Pro ladící účely není nutná implementace metod používaných v CA simulaci. Rozhraní cíle musí obsahovat referenci na paměť, nebo jinou komponentu, na kterou je vázáno.

Paměť by se měla skládat ze tří vrstev podobně jako v Codasip simulátoru. Interní část paměti je zodpovědná za uchování a správu dat a vykonávání operací čtení a zápisu. Tato část tvoří jádro paměti a je neměnná pro všechny případy použití. Lze ji pouze parametrizovat. Nejdůležitějším parametrem paměti je její velikost. Druhou část paměťové komponenty tvoří definice rozhraní. V této části jsou vytvořena rozhraní cílů. Dále se zde nachází informace o latencích a přístupových právech. Tato vrstva paměti se nachází v generované části simulátoru. Poslední vrstvou je hlavní paměťová komponenta, která propojuje obě zmíněné vrstvy.

Paměť obdrží payload od rozhraní cíle. Jejím úkolem je vykonat s daty operaci, kterou payload specifikuje. Paměť by měla fungovat genericky bez ohledu na použitý komunikační protokol. Protokol je zpracován na úrovni rozhraní cíle. Paměť je řízena pouze položkami základního payloadu, které jsou pro všechny protokoly identické.

Hlavní změnou oproti paměti, která byla používána v původním Codasip simulátoru, je interní uspořádání dat. Původní paměť ukládala data jako pole položek o velikosti jednoho slova. Tento přístup vedl k rozsáhlým transformacím při přístupech na nezarovnané adresy nebo při proměnné velikosti dat. Nový návrh paměti ukládá data jako pole položek o velikosti jednoho bajtu. Tento přístup je lépe kompatibilní s přístupem TLM, kde jsou přenášená data ukládána ve stejném formátu.

6.4 Návrh komunikačních protokolů pro sběrnice

Každá sběrnice obsahuje vlastní specifické signály a pravidla pro komunikaci. Tyto informace, popsané ve specifikaci jednotlivých sběrnic, lze použít pro tvorbu nových TLM protokolů. Z pohledu Codasip simulátoru existuje několik důležitých částí, které je potřeba u nového protokolu definovat. Obecně je při tvorbě nových protokolů doporučeno vycházet z doporučeného TLM protokolu. Doporučený protokol definuje payload, fáze komunikace a základní pravidla pro chování obou komunikujících stran.

Prvním důležitým krokem je vytvořit konverzní funkci mezi odpovědí transakce v TLM payloadu a příslušným signálem odpovědi sběrnice. Toto je jediný nutný krok pro implementaci komunikace v IA simulaci.

Dalším krokem je definice fází komunikace a směru posílání dat v jednotlivých fázích. Z pohledu Codasip simulátoru odpovídá každé fázi, použité při volání metody `transport`, jedna fáze TLM protokolu. Z pohledu implementace může být v některých případech nutné definovat další dodatečné fáze. To se týká například situací, kdy v rámci jedné fáze probíhá přenos dat oběma směry. Rozhraní, které odeslalo transakci, nemůže vědět, zda má příjemce připravena validní data pro odpověď. Pokud příjemce transakce nemůže okamžitě poskytnout potřebná data, je povinen je poskytnout později pomocí další pomocné fáze. Pokud je to možné, je vhodné využít již existující fáze definované v knihovně SystemC. Pro většinu podporovaných sběrnic je dostačující převzít základní fáze a případně upravit jejich sémantiku. Pouze pro sběrnici AXI4-lite je nutné definovat dodatečné transportní fáze.

Posledním krokem je definice konverzních funkcí mezi signály sběrnice a položkami payloadu. Všechny konverze musí být obousměrné. Většina základních informací má dedikované položky v základním TLM payloadu. Jedná se například o adresu, data a typ operace. Pro informace, které nemají dedikované položky a musí být přenášeny v rámci transakce, je nutné vytvořit rozšíření payloadu (viz kapitola 5.3.4). V rozšíření je vyhrazená speciální položka pro každou potřebnou informaci. Výhodou tohoto přístupu je vlastní definice konkrétních hodnot. Není potřeba speciálních konverzních funkcí. Nevýhodou je nutnost kontrolovat přítomnost daného rozšíření a vzniklá režie při jeho vytváření a odstraňování.

Při dodržení tohoto postupu není potřeba definovat přesné chování jednotlivých rozhraní iniciátorů a cílů. Tato definice je již dostupná ve specifikaci sběrnic, ze kterých byl nový TLM protokol vytvořen. Rozhraní se musí chovat stejným způsobem jako zařízení sběrnice při daných hodnotách jednotlivých signálů, které jsou odvoditelné z TLM komunikace. Pokud takto vytvořený protokol nespecifikuje některou z potřebných částí definice TLM protokolu, je použita definice z doporučeného TLM protokolu.

Podstatnou změnou oproti doporučenému TLM protokolu je nevyužití parametru zpoždění v transportních metodách. Toto omezení se týká IA i CA simulace a komunikace v obou směrech. Parametr `delay` v transportních metodách TLM má umožňovat iniciátorům předat informaci o případném zpoždění před vykonáním transakce. Cílům umožňuje předávat simulované zpoždění daných operací. Tento koncept zpoždění je natolik odlišný od současného řešení Codasip simulátoru, že ho nelze jednoduchým způsobem aplikovat na nově vytvořené protokoly. V rámci nových protokolů je transakce vykonávána okamžitě bez možnosti zpoždění na obou stranách komunikace. Zpoždění simulující například latenci paměti je vytvořeno časovým rozdílem mezi aktivací jednotlivých transakčních fází.

Signály sběrnice určující hodinový signál a reset jsou v této části záměrně ignorovány. Tikání hodinového signálu je simulováno globálně pro celý simulátor. Toho může být docíleno pomocí volání metody `clock_cycle` nad všemi simulačními komponentami. Toto řešení bylo využito v původní implementaci Codasip simulátoru. Další možností je použít

simulační jádro knihovny SystemC pro automatické posouvání simulačního času, čímž lze simulovat hodinový signál s konkrétní periodou. Reset signály rovněž nejsou reprezentovány v transakcích. Každé rozhraní iniciátora a cíle obsahuje `reset` metodu, která je volána ze simulátoru při daných situacích.

Definice použitého protokolu musí existovat i pro rozhraní typu IA. Zde není nutné definovat fáze a jejich sémantiku, protože při IA simulaci se fáze nepoužívají. Je ale nutné definovat konverzi parametrů transakce z formátu Codasip simulátoru do formátu TLM payloadu. Všechny operace specifikované rozhraním typu IA (viz kapitola 4.3.1) lze převést do položek payloadu jednoduchou konverzí. Metody `dread`, `dwrite` a `load` posílají payload pomocí ladící transportní metody soketu. Ostatní metody využívají metodu soketu pro blokující transport. Metody `read` a `dread` nastavují příkazovou položku payloadu na hodnotu `tlm::TLM_READ_COMMAND`. Metody `write`, `dwrite` a `load` nastavují příkazovou položku payloadu na hodnotu `tlm::TLM_WRITE_COMMAND`. Adresa je přiřazena do payloadu bez nutnosti konverze. Parametr `response` je konvertován na základě použitého komunikačního protokolu. Parametr `count` určující velikost dat v bajtech je konvertován na velikost datového pole v payloadu. Parametr `data` u operací zápisu je konvertován do datového pole payloadu.

Jedním ze základních parametrů každé operace jsou přenášená data. Pro operace čtení jsou data přenášena od cíle k iniciátorovi. Pro operaci zápisu jsou přenášena od iniciátora k cíli. Rozhraní iniciátora musí zajistit korektní konverzi dat pro transportní metody typu IA i CA. Simulátor procesoru vyžaduje data ve formátu jedné celočíselné hodnoty o dostatečné bitové šířce. Payload a ostatní komponenty paměťového subsystému vyžadují data ve formátu bajtového pole. Rozhraní iniciátora musí být schopné vykonat konverzi dat mezi oběma formáty. Při konverzi dat je nutné zohlednit endianitu použitého rozhraní iniciátora. V případě endianity typu *little* mohou být data přímo zkopírována do datového pole, například pomocí funkce `memcpy`. V případě endianity typu *big* je potřeba navíc provést přehození bajtů pole. Konverze stejných dat mají tedy jiný výsledek při použití různého typu endianity. Protože jsou data v ostatních komponentách paměťového subsystému reprezentována ve stejném formátu, nejsou potřeba další datové konverze.

6.4.1 Protokol pro sběrnici AHB3-lite

Následující signály sběrnice AHB3-lite jsou konvertovány do příslušných položek TLM payloadu:

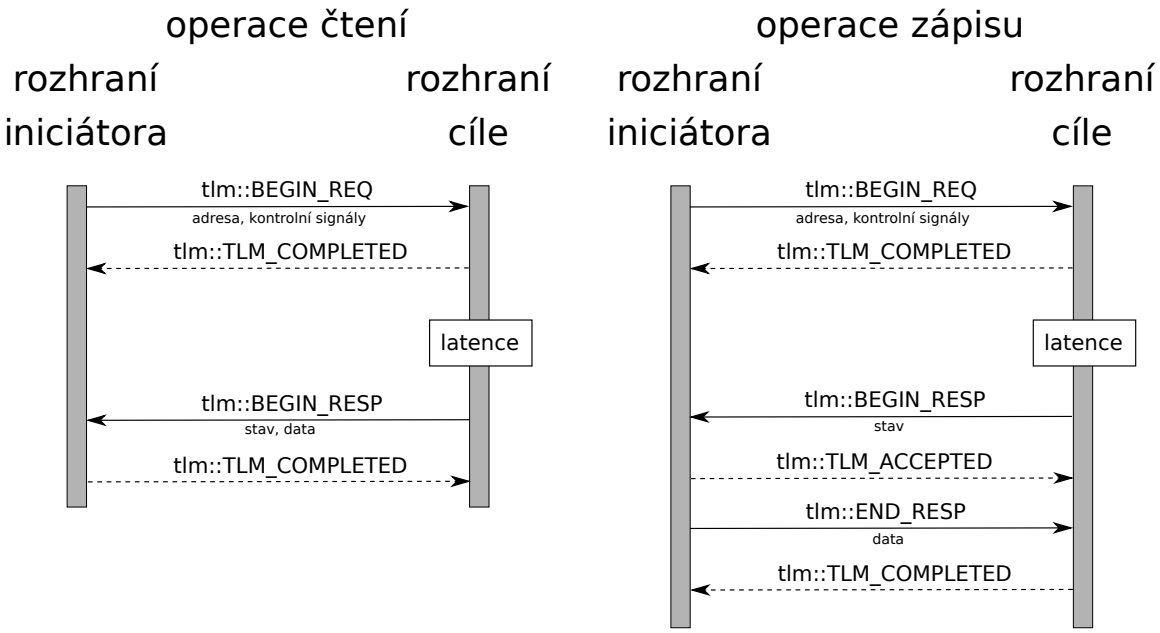
- **HADDR**: konvertováno beze změn do adresové položky payloadu (`m_address`).
- **HSIZE**: konvertováno do bajtové masky payloadu (`m_byte_enable`). Podle hodnoty signálu je určena velikost dat v bajtech. Tento počet bajtů je v masce povolen a ostatní bajty jsou ignorovány.
- **HWDATA** a **HRDATA**: konvertováno do datové položky payloadu (`m_data`). Konverze mezi celočíselnou hodnotou a polem bajtů probíhá podle postupu popsaného v kapitole 6.4.

- **HWRITE**: konvertováno do příkazové položky payloadu (`m_command`). Logická hodnota jedna je konvertována na příkaz `tlm::TLM_WRITE_COMMAND`. Logická hodnota nula je konvertována na příkaz `tlm::TLM_READ_COMMAND`. Tato konverze je provedena, pouze pokud signál `HTRANS` nabývá hodnot `NONSEQ` nebo `SEQ`. V opačném případě je příkaz nastaven na hodnotu `tlm::TLM_IGNORE_COMMAND` bez ohledu na hodnotu signálu `HWRITE`.

- **HRESP**: konvertováno do odpovědní položky payloadu (`m_response_status`). Hodnota logické nuly je reprezentována odpovědí `tlm::TLM_OK_RESPONSE`. Hodnota logické jedničky je reprezentována jakoukoliv zápornou hodnotou odpovědi, například `tlm::TLM_GENERIC_ERROR_RESPONSE`.

Ostatní signály sběrnice jsou reprezentovány v rozšíření payloadu. Jedná se o signály `HBURST`, `HMASTLOCK` , `HPROT`, `HTRANS` a `HREADYOUT`. Hodnota signálu `HSIZE` je také přítomna v rozšíření, přestože byla použita pro vytvoření datové masky. Konverze odpovědi pro IA simulaci se řídí stejnými pravidly jako konverze signálu `HRESP`.

Protokol reprezentující sběrnici AHB3-lite používá tři transakční fáze převzaté z doporučeného TLM protokolu. Typické použití fází pro operace čtení a zápisu je znázorněno na obrázku 6.1.



Obrázek 6.1: Schéma použití fází protokolu AHB3-lite.

- **tlm::BEGIN_REQ**: tuto fázi zahajuje rozhraní iniciátora. Jsou v ní obsaženy signály sběrnice nastavované rozhraním typu master, s výjimkou dat posílaných při operaci zápisu. Rozhraní cíle je povinno v každé situaci tato data náležitě zpracovat. V rámci této fáze nejsou posílány zpět do rozhraní iniciátora žádné informace a není potřeba využití fáze `tlm::END_REQ` z doporučeného TLM protokolu.

- **tlm::BEGIN_RESP**: tato fáze je zahájena rozhraním cíle v situaci, kdy cíl zpracovává operaci zadanou fází `tlm::BEGIN_REQ` a uplyne simulovaná latence paměti. Účelem této fáze je dokončit zahájenou operaci. Rozhraní cíle nastavuje v této fázi hodnoty signálů `HRESP`, `HREADYOUT` a v případě operace čtení hodnotu signálu `HRDATA`. V případě operace zápisu je rozhraní iniciátora povinno poskytnout validní hodnotu signálu `HWDATA`. Pokud tato fáze není vyvolána v rámci hodinového cyklu, rozhraní iniciátora předpokládá, že cílové rozhraní nastavilo signály sběrnice `HRESP` a `HREADYOUT` na hodnotu logické nuly. Pouze pokud rozhraní iniciátora neeviduje žádnou nedokončenou operaci, je předpokládaná hodnota signálu `HREADYOUT` v logické jedničce.

- **tlm::END_RESP**: Pokud rozhraní iniciátora aktuálně nemůže poskytnout zapisovaná data vyžadována ve fázi `tlm::BEGIN_RESP`, je povinno indikovat tuto skutečnost návratovou hodnotou `tlm::TLM_ACCEPTED`. Následně musí zahájit tuto fázi ve chvíli, kdy bude mít data k dispozici. Pro potřeby Codasip simulátoru musí být data k dispozici ve stejném hodinovém cyklu, jako byla provedena fáze `tlm::BEGIN_RESP`. V rámci této fáze je validní pouze signál `HWDATA`.

6.4.2 Protokoly pro sběrnice CPB a CPB-lite

Následující signály sběrnic CPB a CPB-lite jsou konvertovány do příslušných položek TLM payloadu:

- **ADDR**: konvertováno beze změn do adresové položky payloadu (`m_address`).

- **WSTRB**: konvertováno do bajtové masky payloadu (`m_byte_enable`). Každý bit signálu je konvertován na jeden bajt masky o hodnotě `0x0` pro bit nula nebo `0xff` pro bit jedna.

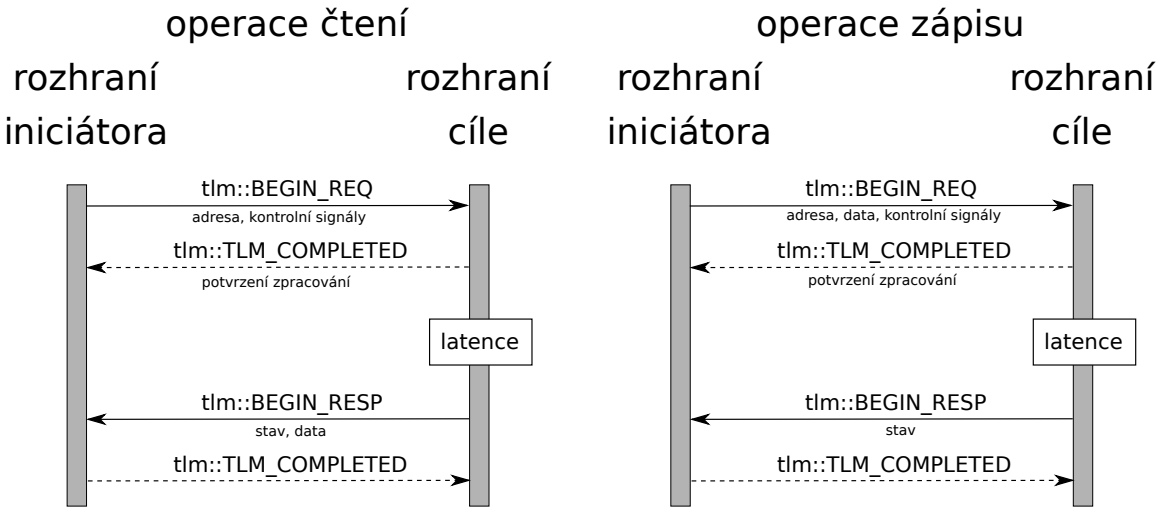
- **WDATA** a **RDATA**: konvertováno do datové položky payloadu (`m_data`). Konverze mezi celočíselnou hodnotou a polem bajtů probíhá podle postupu popsaného v kapitole 6.4.

- **WRITE**: konvertováno do příkazové položky payloadu (`m_command`). Logická hodnota jedna je konvertována na příkaz `tlm::TLM_WRITE_COMMAND`. Logická hodnota nula je konvertována na příkaz `tlm::TLM_READ_COMMAND`.

- **RESP**: konvertováno do odpovědní položky payloadu (`m_response_status`). Hodnota logické nuly je reprezentována odpovědí `tlm::TLM_OK_RESPONSE`. Hodnota logické jedničky je reprezentována jakoukoliv zápornou hodnotou odpovědi, například `tlm::TLM_GENERIC_ERROR_RESPONSE`.

Signál `AREADY` sběrnice CPB je reprezentován v rozšíření payloadu. Sběrnice CPB-lite nemá žádné signály reprezentované v rozšíření. Signály `AVALID` a `VALID` nejsou reprezentovány jako uložené informace v payloadu. Jejich hodnota je závislá na vyvolání konkrétní fáze komunikace. Pokud příslušná fáze není vykonána, předpokládá se hodnota signálu v logické nule, v opačném případě v logické jedničce. Toto zjednodušení je možné díky neplatnosti ostatních signálů dané fáze, pokud je hodnota signálu `AVALID` nebo `VALID` v logické nule. Konverze odpovědi pro IA simulaci se řídí stejnými pravidly jako konverze signálu `RESP`.

Protokoly reprezentující sběrnice CPB a CPB-lite používají dvě transakční fáze převzaté z doporučeného TLM protokolu. Typické použití fází pro operace čtení a zápisu je znázorněno na obrázku 6.2.



Obrázek 6.2: Schéma použití fází protokolů CPB a CPB-lite.

- **tlm::BEGIN_REQ**: tuto fázi zahajuje rozhraní iniciátora a jsou v ní obsaženy všechny signály sběrnice nastavované rozhraním typu master. Pokud aktuální transakce provádí operaci čtení, jsou signály `WDATA` a `WSTRB` ignorovány. Zahájení této fáze znamená hodnotu signálu `AVALID` v logické jedničce. Rozhraní cíle je povinno informace z této transakce náležitě zpracovat. Případě sběrnice CPB je možné indikovat neschopnost zpracování dat signálem `AREADY`. Kromě hodnoty signálu `AREADY`, která nesmí být závislá na aktuální transakci, nejsou posílány zpět do rozhraní iniciátora žádné informace a není potřeba využití fáze `tlm::END_REQ` z doporučeného TLM protokolu.
- **tlm::BEGIN_RESP**: tato fáze je zahájena rozhraním cíle v situaci, kdy cíl zpracovává operaci zadanou fází `tlm::BEGIN_REQ` a uplyne simulovaná latence paměti. Účelem této fáze je dokončit zahájenou operaci. Rozhraní cíle nastavuje v této fázi hodnotu signálu `RESP` a v případě operace čtení hodnotu signálu `RDATA`. Zahájení této fáze znamená hodnotu signálu `VALID` v logické jedničce. Pokud je aktuální hodnota tohoto signálu v logické nule, tato fáze neproběhne. V rámci této fáze nejsou posílány zpět do rozhraní cíle žádné informace a není potřeba využití fáze `tlm::END_RESP` z doporučeného TLM protokolu.

6.4.3 Protokol pro sběrnici AXI4-lite

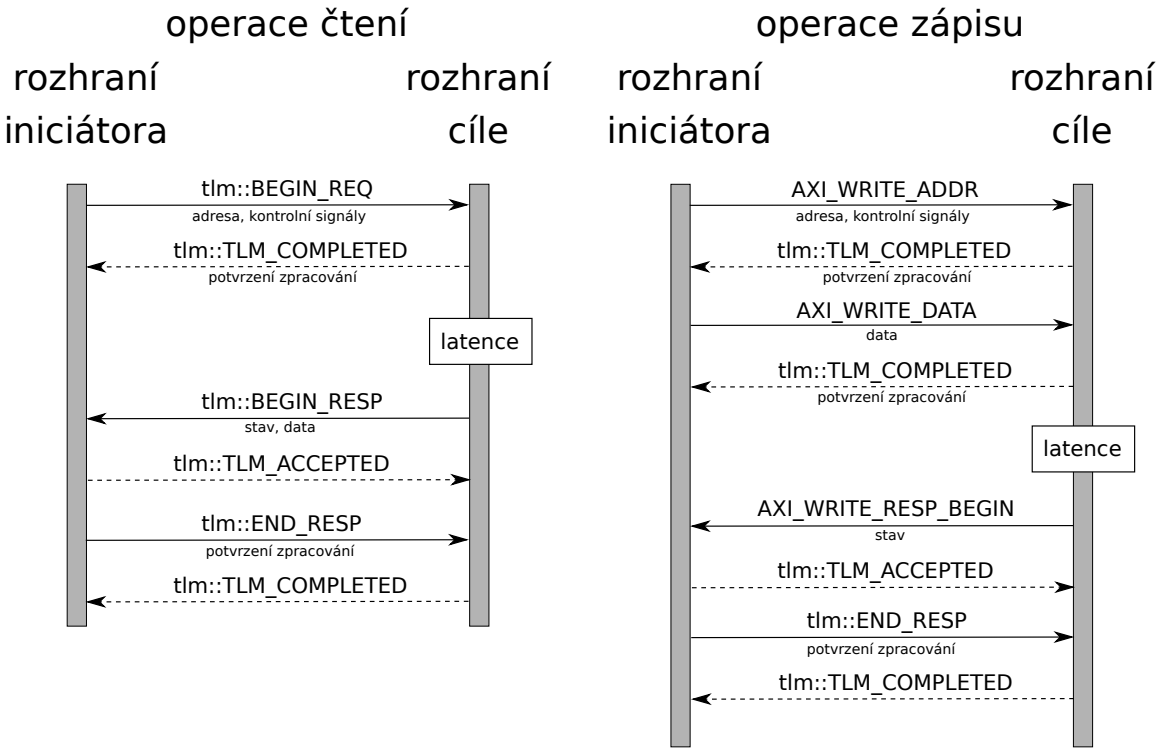
Následující signály sběrnice AXI4-lite jsou konvertovány do příslušných položek TLM payloadu:

- **AWADDR** a **ARADDR**: konvertováno beze změn do adresové položky payloadu (`m_address`).
- **WSTRB**: konvertováno do bajtové masky payloadu (`m_byte_enable`). Každý bit tohoto signálu je konvertován na jeden bajt masky o hodnotě `0x0` pro bit nula nebo `0xff` pro bit jedna.
- **WDATA** a **RDATA**: konvertováno do datové položky payloadu (`m_data`). Konverze mezi celočíselnou hodnotou a polem bajtů probíhá podle postupu popsaného v kapitole 6.4.

- **BRESP** a **RRESP**: konvertováno do odpovědní položky payloadu (`m_response_status`). Hodnota nula tohoto signálu je reprezentována odpovědí `tlm::TLM_OK_RESPONSE`. Hodnota dva je reprezentována jakoukoliv zápornou hodnotou odpovědi, například `tlm::TLM_GENERIC_ERROR_RESPONSE`. Ostatní hodnoty těchto signálů nejsou v protokolu podporovány.

Signály `AWPROT`, `ARPROT`, `AWREADY`, `WREADY`, `BREADY`, `ARREADY` a `RREADY` sběrnice AXI4-lite jsou reprezentovány v rozšíření payloadu. Signály `AWVALID`, `WVALID`, `BVALID`, `ARVALID` a `RVALID` nejsou reprezentovány jako uložené informace v payloadu. Jejich hodnota je závislá na vyvolání konkrétní fáze komunikace. Pokud příslušná fáze není vykonána, předpokládá se hodnota signálu v logické nule, v opačném případě v logické jedničce. Toto zjednodušení je možné díky neplatnosti ostatních signálů dané fáze, pokud je hodnota příslušného `VALID` signálu v logické nule. Konverze odpovědi pro IA simulaci se řídí stejnými pravidly jako konverze signálů `BRESP` a `RRESP`.

Pro protokol reprezentující sběrnici AXI4-lite nestačí použití standardních fází a je potřeba definovat nové fáze. Každá fáze tohoto protokolu by měla odpovídat jednomu komunikačnímu kanálu sběrnice. To znamená, že některé signály sběrnice z různých komunikačních kanálů budou sdílet datový prostor v payloadu. Jedná se především o signály typu `READY` pro jednotlivé kanály. Typické použití fází pro operace čtení a zápisu je znázorněno na obrázku 6.3.



Obrázek 6.3: Schéma použití fází protokolu AXI4-lite.

- **tlm::BEGIN_REQ**: tuto fázi zahajuje rozhraní iniciátora a reprezentuje komunikační kanál `read address`. Obsahuje hodnoty signálů `ARADDR` a `ARPROT`. Zahájení této fáze znamená hodnotu signálu `ARVALID` v logické jedničce. Rozhraní cíle je povinno informace z této transakce náležitě zpracovat nebo indikovat neschopnost zpracování signálem `AREADY`. Kromě hodnoty signálu `AREADY`, která nesmí být závislá na aktuální transakci, nejsou posílány zpět do rozhraní iniciátora žádné informace a není potřeba využití fáze `tlm::END_REQ` z doporučeného TLM protokolu.

- **tlm::BEGIN_RESP**: tato fáze je zahájena rozhraním cíle v situaci, kdy cíl zpracovává operaci čtení zadanou fází `tlm::BEGIN_REQ` a uplyne simulovaná latence paměti. Jedná se o reprezentaci kanálu `read data`. Fáze obsahuje hodnoty signálů `RDATA` a `RRESP`. Zahájení této fáze znamená hodnotu signálu `RVALID` v logické jedničce. Tato fáze je iniciátorem zpracována pouze, pokud je aktuální hodnota signálu `RREADY` v logické jedničce. Kvůli indikaci zpracování této fáze musí rozhraní iniciátora poskytnout validní hodnotu signálu nebo použít pomocnou fázi v případě, že hodnota není aktuálně k dispozici.

- **tlm::END_RESP**: pokud rozhraní iniciátora aktuálně nemůže poskytnout informace o zpracování fáze `tlm::BEGIN_RESP`, je povinno indikovat tuto skutečnost návratovou hodnotou `tlm::TLM_ACCEPTED`. Následně musí zahájit tuto fázi ve chvíli, kdy bude mít danou informaci k dispozici. Pro potřeby Codasip simulátoru musí být tato fáze vyvolána ve stejném hodinovém cyklu, jako byla provedena fáze `tlm::BEGIN_RESP`. V rámci této fáze je validní pouze signál `RREADY`.

- **AXI_WRITE_ADDR**: tuto fázi zahajuje rozhraní iniciátora a reprezentuje komunikační kanál `write address`. Obsahuje hodnoty signálů `AWADDR` a `AWPROT`. Zahájení této fáze znamená hodnotu signálu `AWVALID` v logické jedničce. Rozhraní cíle je povinno tuto fázi náležitě zpracovat nebo indikovat neschopnost zpracování signálem `AWREADY`. Kromě hodnoty signálu `AWEADY`, která nesmí být závislá na aktuální transakci, nejsou posílány zpět do rozhraní iniciátora žádné informace a není potřeba využití další pomocné fáze.

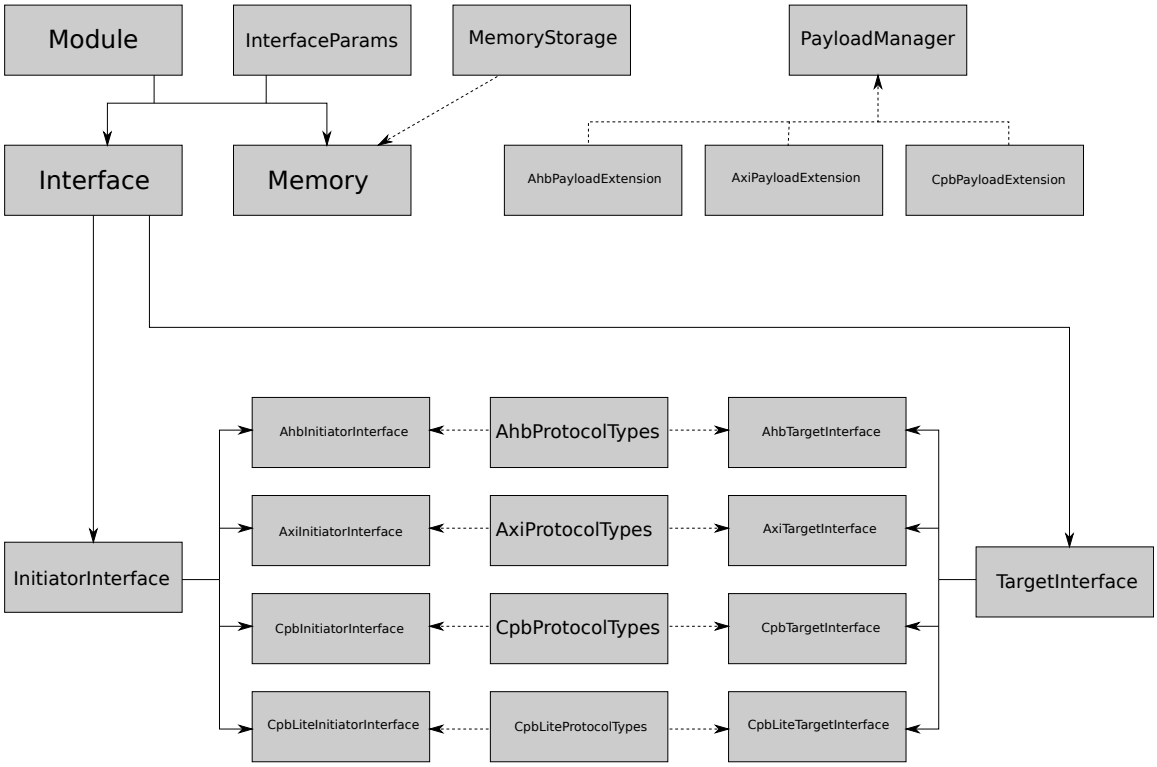
- **AXI_WRITE_DATA**: tuto fázi zahajuje rozhraní iniciátora a reprezentuje komunikační kanál `write data`. Obsahuje hodnoty signálů `WDATA` a `WSTRB`. Zahájení této fáze znamená hodnotu signálu `WVALID` v logické jedničce. Rozhraní cíle je povinno tuto fázi náležitě zpracovat nebo indikovat neschopnost zpracování signálem `WREADY`. Kromě hodnoty signálu `WREADY`, která nesmí být závislá na aktuální transakci, nejsou posílány zpět do rozhraní iniciátora žádné informace a není potřeba využití další pomocné fáze.

- **AXI_WRITE_RESP_BEGIN**: tato fáze je zahájena rozhraním cíle v situaci, kdy cíl zpracovává operaci zápisu zadanou fázemi `AXI_WRITE_ADDR` a `AXI_WRITE_DATA` a uplyne simulovaná latence paměti. Jedná se o reprezentaci kanálu `write response`. Fáze obsahuje hodnotu signálu `BRESP`. Zahájení této fáze znamená hodnotu signálu `BVALID` v logické jedničce. Tato fáze je iniciátorem zpracována pouze, pokud je aktuální hodnota signálu `BREADY` v logické jedničce. Kvůli indikaci zpracování této fáze musí rozhraní iniciátora poskytnout validní hodnotu signálu nebo použít pomocnou fázi v případě, že hodnota není aktuálně k dispozici.
- **AXI_WRITE_RESP_END**: pokud rozhraní iniciátora aktuálně nemůže poskytnout informace o zpracování fáze `AXI_WRITE_RESP_BEGIN`, je povinno indikovat tuto skutečnost návratovou hodnotou `tlm::TLM_ACCEPTED`. Následně musí zahájit tuto fázi ve chvíli, kdy bude mít danou informaci k dispozici. Pro potřeby Codasip simulátoru musí být tato fáze vyvolána ve stejném hodinovém cyklu, jako byla provedena fáze `AXI_WRITE_RESP_BEGIN`. V rámci této fáze je validní pouze signál `BREADY`.

Kapitola 7

Implementace řešení

Při implementaci řešení bylo využito stávajících konstrukcí Codasip simulátoru a konstrukcí definovaných v knihovně SystemC. Výsledné řešení vychází z návrhu zpracovaného v kapitole 6. Implementace byla provedena v jazyce C++ za použití objektově orientovaného přístupu, dědičnosti a šablonování. Implementováno bylo několik hlavních tříd reprezentujících součásti a komponenty paměťového subsystému. Dále byly implementovány pomocné konstrukce pro správnou funkčnost konečného řešení. Graf implementovaných tříd a jejich propojení lze vidět na obrázku 7.1.



Obrázek 7.1: Diagram nově implementovaných tříd paměťového subsystému. Plná čára značí dědičnost, přerušovaná čára značí přímé použití nebo instanciaci.

Implementace nových komponent je přítomna na přiloženém paměťovém médiu (viz B).

7.1 Událostmi řízená simulace

Jedním z největších implementačních problémů této práce bylo zakomponovat existující součásti Codasip simulátoru do SystemC simulace. Simulační jádro knihovny SystemC pracuje na principu událostmi řízené simulace (viz kapitola 5.2). Codasip simulátor pracuje na principu sekvenční simulace. Rozdíly v těchto dvou principech musí být vyřešeny na rozhraní mezi původním Codasip simulátorem a novými komponentami. Abychom nemuseli provádět rozsáhlé změny při integraci, byla synchronizace mezi těmito přístupy umístěna do implementace rozhraní iniciátorů. Zmíněný problém se netýká IA simulace, protože každá operace je reprezentována právě jedním transportním voláním. Provádění operace v IA simulaci je okamžité a jednorázové. Nejsou tedy v průběhu simulace přítomny události z paměťového subsystému a simulace je řízena stejným sekvenčním způsobem jako v původním Codasip simulátoru.

V rámci CA simulace volá jádro procesoru transportní metody rozhraní iniciátora. Tyto metody se typicky volají pravidelně každý hodinový cyklus, aby se na simulované sběrnici nacházely stále aktuální hodnoty. TLM posílá transakce pouze ve specifických fázích a s validními informacemi. V některých situacích, například nevalidní data při čekání na latenci paměti, nejsou aktivovány v hodinovém cyklu všechny fáze operace. Transportní metoda rozhraní iniciátora je ale i v této situaci zavolána a procesor očekává aktuální hodnoty signálů na sběrnici. Rozhraní iniciátora musí být schopné odvodit signály sběrnice i v případě neexistující transakce od rozhraní cíle. Toto odvození je specifikováno pro konkrétní sběrnice v rámci jejího komunikačního protokolu (viz kapitola 6.4). Z pohledu SystemC simulace je rozhraní procesoru i paměti aktivováno ve stejném simulačním čase a není možné určit pořadí aktivace jednotlivých procesů. V okamžiku, kdy rozhraní iniciátora kontroluje přijetí požadované fáze, je nemožné zjistit, zda rozhraní cíle odešle danou transakci ve stejném hodinovém cyklu nebo v tomto cyklu požadovaná komunikace neproběhne.

Tento problém má několik možných řešení. Ne všechna řešení jsou vhodná pro použití v této práci. V následujícím seznamu jsou uvedeny vyzkoušené způsoby řešení tohoto problému. U každého z nich je uveden důvod, proč nebyl zvolen pro finální implementaci.

- Posílání transakcí v každém hodinovém cyklu: toto prvotní řešení vychází z původní sekvenční implementace Codasip simulátoru. V tomto scénáři jsou procesor i paměť nuceny v každém hodinovém cyklu provést všechny fáze komunikace bez ohledu na aktuální stav komponent. Rozhraní se v takovém případě mohou spolehnout na příjem požadovaných typů transakcí a mohou čekat libovolně dlouhou dobu na jejich doručení. Tato metoda nebyla použita z několika důvodů. Především se jedná o nemožnost detekce komunikační chyby kvůli potenciálně nekonečně dlouhému čekání. Dalším důvodem je zpomalení simulace z důvodu posílání velkého množství transakcí. Toto řešení také není blízké doporučenému TLM protokolu, kde se posílají pouze transakce obsahující užitečné informace.

- Použití metody `sc_core::sc_time_to_pending_activity`: tato metoda knihovny SystemC vrací simulační čas pro vykonání další simulační události. Této informace lze využít k určení toho, zda některá z komponent simulace kromě procesoru bude vykonávat operace v aktuálním hodinovém cyklu. Při kontrole přijetí transakce na straně procesoru, v rámci rozhraní iniciátora, lze přesně určit, zda v tomto cyklu transakce dorazí. Tato metoda má jednu podstatnou nevýhodu, kvůli které nebyla použita. Řešení je funkční pouze v případě, že je v simulaci přítomen pouze jeden proces, který provádí synchronizaci mezi sekvenční a událostmi řízenou simulací. Po-

kud by například simulace obsahovala dva nezávislé procesory, zmíněná metoda by detekovala tyto komponenty jako aktivní v aktuálním hodinovém cyklu, což by znemožnilo detekci příchozích transakcí. Knihovna SystemC nemá k dispozici metody pro kontrolu aktivity jednotlivých procesů.

- Různé hodinové cykly pro komponenty: myšlenkou této metody je aktivace procesoru v jiném simulačním čase než aktivace ostatních komponent simulace. Jednou variantou je simulační časy komponent nepatrně posunout. Další možností je rozdělit simulační čas hodinového cyklu na sekce a aktivaci komponent střídat. Obě tyto možnosti by znamenaly rozsáhlé zásahy do implementace paměti a jiných komponent. Především latence by musela být pozměněna oproti specifikaci, aby reflektovala rozdíl simulačních časů jednotlivých komponent. Tento problém by se potenciálně mohl rozšiřovat při zapojení dodatečných komponent na cestě mezi procesorem a pamětí. Také výpisy ze simulace by byly hůře uživatelsky čitelné z důvodu nesouhlasících časových značek.

- Předčasná specifikace latence: toto řešení využívá parametr `delay` v TLM transakcích, případně rozšíření payloadu s podobnou funkcionalitou. Cílová komponenta transakce by byla povinna vyplnit tento atribut hodnotou simulačního času, ve kterém bude posílat další fázi dané operace, pokud je tato fáze zahájena cílem. Komponenta tedy musí předem znát latenci prováděné operace a případně latenci dalších komponent na datové cestě. V případě propojení pouze procesoru a paměti je toto řešení použitelné. Některé komponenty, které mohou být potenciálně implementovány v rámci simulátoru, nemusí vždy předem znát latenci konkrétní operace. Příkladem takovéto komponenty je vyrovnávací paměť. Toto řešení bylo zamítnuto, z důvodu možných kompilací s budoucí implementací komponent.

Nejvhodnější způsob řešení popsaného problému je využití mechanismu delta cyklů v SystemC simulaci. Základní myšlenkou tohoto řešení je čekání iniciátora na příslušnou transakci v rozmezí několika delta cyklů. Po uplynutí předem zadaného počtu delta cyklů předpokládá iniciátor, že transakce v daném hodinovém cyklu nedorazí. Pokud transakce dorazí v rámci stejného hodinového cyklu, ale po uplynutí daného počtu delta cyklů, jedná se o chybu simulátoru. Důležitou otázkou, kterou je třeba při použití tohoto řešení zodpovědět, je, jak určit počet delta cyklů pro čekání iniciátora. Tento údaj závisí na implementaci jednotlivých komponent na cestě mezi iniciátorem a cílem. Každá komponenta má na základě implementace předem daný maximální počet delta cyklů nutných ke zpracování každé transakce. V případě této práce se zabýváme pouze přímou komunikací mezi procesorem a pamětí. Jediná komponenta, na které závisí počet delta cyklů, je cílové rozhraní paměti. Na základě implementace připojeného rozhraní cíle lze v této práci přesně určit maximální počet delta cyklů pro rozhraní iniciátora. Pro budoucí rozšíření této práce je počet delta cyklů parametrem všech rozhraní iniciátorů a může být dynamicky konfigurován pomocí Codasip nástrojů.

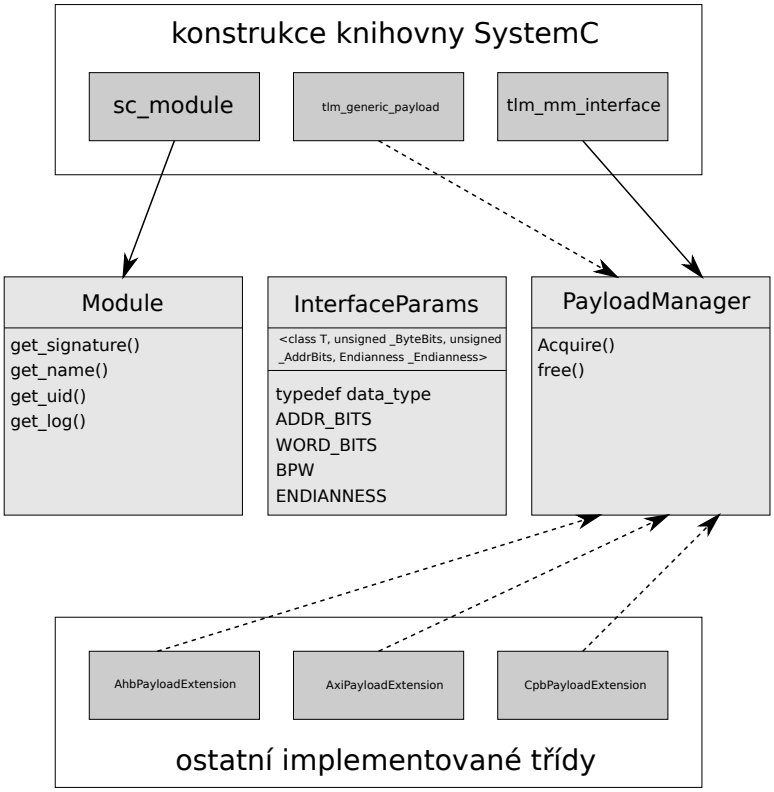
Cílová rozhraní podporovaných komunikačních protokolů jsou implementována s hodnotou maximálního počtu delta cyklů jedna. Veškeré transakce ze strany rozhraní cíle jsou odeslány v rámci prvního delta cyklu. Rozhraní iniciátora na straně procesoru mohou čekat maximálně jeden delta cyklus na doručení všech požadovaných fází.

7.2 Implementace generických komponent

Tato část práce se zabývá částmi paměťového subsystému, které nejsou závislé na konkrétním komunikačním protokolu. Především do této kategorie patří obecná rozhraní iniciátora a cíle primárně určená pro komunikaci v IA simulaci. Dále se jedná o implementaci paměti a pomocných struktur a funkcí.

7.2.1 Pomocné konstrukce

Pro správnou funkčnost paměťového subsystému je potřeba celá řada podpůrných konstrukcí, které jsou implementovány mimo hlavní simulační komponenty. Jak bylo řečeno v kapitole 6, rozhraní i paměti mají některé společné vlastnosti. Ty je vhodné vyčlenit do speciálních tříd za účelem unifikace funkcionality a zjednodušení zdrojového kódu. Implementované pomocné třídy a jejich hlavní položky jsou zobrazeny na obrázku 7.2.



Obrázek 7.2: Diagram implementovaných pomocných konstrukcí. Plná čára značí dědičnost, přerušovaná čára značí použití.

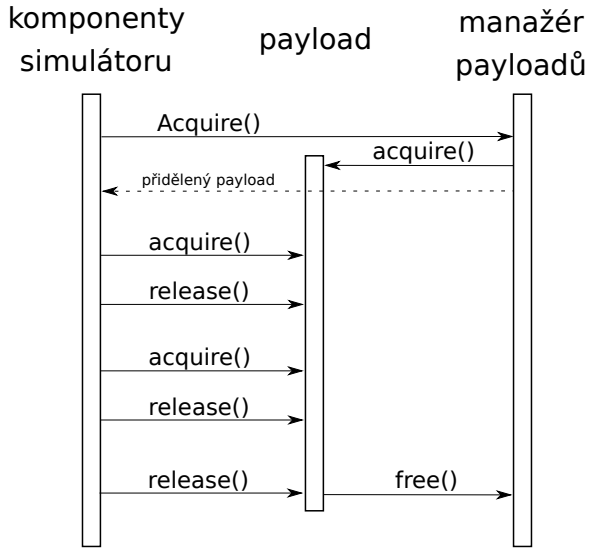
Třída `Module` byla vytvořena za účelem sjednotit funkcionalitu spojenou s moduly knihovny SystemC. Rovněž obsahuje základní metody komponent Codasip simulátoru. Tyto metody jsou nutné pro správnou integraci a použití v rámci simulátoru. Tato třída dědí od třídy `sc_core::sc_module` z knihovny SystemC a obsahuje jméno, signaturu a jednoznačný identifikátor komponenty v rámci Codasip simulátoru. Implementace veškerých simulačních komponent by měla dědit z této třídy.

Třída `InterfaceParams` sdružuje hlavní parametry všech komunikačních kanálů. Jedná se o statickou třídu, která obsahuje pouze definice datových typů a konstant. Všechny tyto

informace třída získá pomocí šablony použité při její konstrukci. Parametry této šablony jsou následující:

- datový typ pro reprezentaci dat
- velikost nejmenší adresovatelné jednotky v bitech
- velikost adresy v bitech
- endianita přenášených dat

Tyto informace jsou automaticky generované z popisu komponenty v jazyce CodAL při vytváření simulátoru. Na základě vstupních informací jsou definovány další konstanty a datové typy. Jedná se například o datový typ pro efektivní předávání dat mezi volanými funkcemi. Dalším příkladem je velikost přenášených dat v bitech i v nejmenších adresovatelných jednotkách. Jediná podporovaná velikost nejmenší adresovatelné jednotky je pro účely této práce osm bitů. Jiné velikosti nejsou podporovány z důvodu problémů s konverzí na bajtové pole TLM payloadu. Jednotlivá rozhraní a paměti jsou parametrizovány třídou `InterfaceParams` za použití šablonování. Rozhraní a paměti dědí od této třídy, aby měly přímý přístup k obsaženým informacím.



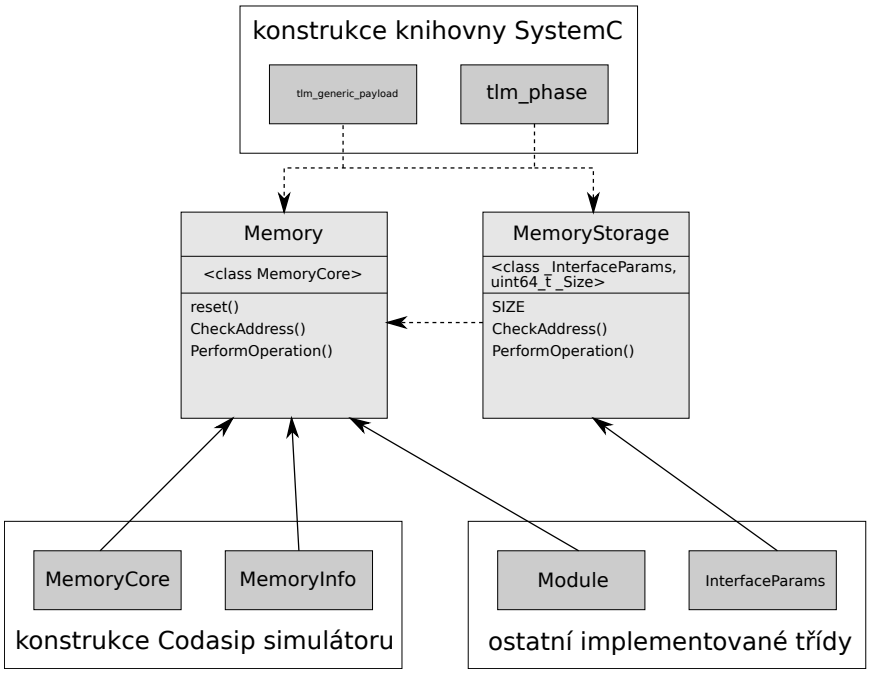
Obrázek 7.3: Scháma použití manažera payloadů.

Třída `PayloadManager` je zodpovědná za správu paměti, kterou zabírají payloady různých typů. Je odvozena od třídy `tlm::tlm_mm_interface` z knihovny SystemC. manažer payloadů je nutný pro rychlý chod simulace. Pokud by neexistoval, musel by každý payload pro novou transakci být znovu alokován v paměti a příslušně nastaven. manažer payloadů umožňuje existenci pouze několika instancí payloadu v průběhu celé simulace. Toho je dosaženo pomocí znovupoužití nepotřebných payloadů namísto jejich odstranění. manažer rozlišuje payloady podle tří kategorií, kterými jsou velikost přenášených dat, použitý protokol a existence datové masky. Každý průnik těchto kategorií má k dispozici vlastní množinu payloadů, která je zvětšována podle potřeby simulace. Pokaždé, když rozhraní potřebuje nový payload, je zavolána metoda `Acquire` třídy `PayloadManager`. Při návratu z této metody dostane rozhraní k dispozici jeden z množiny payloadů daného typu. Zároveň

je inkrementováno počítadlo referencí na daný payload objekt. Hodnoty tohoto počítadla lze měnit i později metodami payloadu `acquire` pro inkrementaci a `release` pro dekrementaci. Pokud počítadlo referencí dosáhne nuly, je automaticky zavolána metoda `free` ze třídy `PayloadManager`. Tato metoda je zodpovědná za nastavení výchozích hodnot payloadu a navrácení objektu zpět do příslušné množiny k dalšímu použití. Počítadlo referencí může být vhodné například pro použití stejného payloadu se všemi potřebnými informacemi ve více fázích komunikace. Grafické zobrazení příkladu použití třídy `PayloadManager` je na obrázku 7.3.

7.2.2 Paměť

Implementace paměti se podobně jako v Codasip simulátoru skládá ze tří částí: hlavní komponenty, generované definice rozhraní a interního paměťového úložiště. Tato myšlenka rozdělení implementace paměti byla zachována. Implementované třídy reprezentující paměť a jejich hlavní položky jsou zobrazeny na obrázku 7.4.



Obrázek 7.4: Diagram implementovaných paměťových konstrukcí. Plná čára značí dědičnost, přerušovaná čára značí použití.

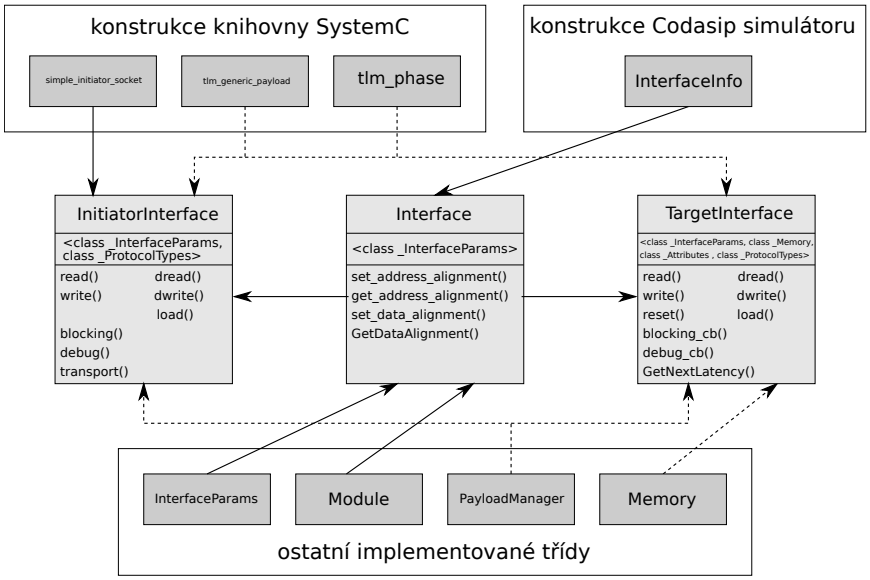
Třída `Memory` reprezentuje hlavní paměťovou komponentu simulátoru. Její rozhraní tvoří především metody `CheckAddress` a `PerformOperation`, které slouží pro přeposílání transakcí do interního paměťového úložiště k dalšímu zpracování. Dále tato třída implementuje metody pro přístup k parametrům paměti, které využívá Codasip simulátor k získávání informací o komponentách. Třída `Memory` dědí od třídy `Module`, aby mohla být validní simulační komponentou. Dále dědí z generované šablonové třídy `MemoryCore`, která určuje rozhraní a interní strukturu paměti.

Třída `MemoryStorage` reprezentuje interní paměťové úložiště, které je použito ve třídě `Memory` a definováno v generované třídě `MemoryCore`. `MemoryStorage` dědí od šablonové třídy `InterfaceParams`, která poskytuje paměti parametry podobně jako třídám reprezen-

tující rozhraní. Zároveň má šablona této třídy další parametr, který určuje velikost paměti v bajtech. To umožňuje konstruktoru této třídy alokovat dostatečné místo pro celý simulovaný paměťový prostor. Další možností je použití řídkého pole pro simulaci paměti o velkých rozměrech. Nejdůležitějšími metodami této třídy jsou `CheckAddress` a `PerformOperation`, které jsou volány z nadřazené paměťové komponenty. První z těchto metod provádí kontrolu rozsahu adresy paměťové operace. Druhá z těchto metod provádí navíc paměťové operace čtení a zápisu. Paměťová operace spočívá v kopírování dat mezi interní pamětí a datovou sekcí payloadu. Kopie dat je přímočará, protože paměť i datová položka payloadu ukládají data ve stejném formátu.

7.2.3 Rozhraní

Obecná rozhraní jsou implementována pomocí tříd `Interface`, `InitiatorInterface` a `TargetInterface`. Tyto třídy a jejich hlavní položky jsou zobrazeny na obrázku 7.5.



Obrázek 7.5: Diagram implementovaných konstrukcí obecných rozhraní. Plná čára značí dědičnost, přerušovaná čára značí použití.

Třída `Interface` implementuje základní funkcionalitu společnou pro všechny typy rozhraní. Především dědí od třídy `Module`, která jí poskytuje vlastnosti simulační komponenty. Dále dědí od šablonové třídy `InterfaceParams`, která poskytuje přístup k základním parametrům daného rozhraní. Podobně jako ve třídě `Memory` jsou zde implementovány metody umožňující simulátoru získávání podrobných informací o instanci rozhraní. V definici rozhraní jsou přítomny položky pro nastavení zarovnání adresy a dat společně s přístupovými metodami k těmto informacím. Hodnoty těchto položek jsou přítomny v generované části simulátoru a jsou převzaty z definice rozhraní v jazyce CodAL (viz kapitola 4.3.3).

Třída `InitiatorInterface` je rozšířením třídy `Interface` a obsahuje společnou funkcionalitu pro rozhraní iniciátorů všech komunikačních protokolů. Především se jedná o implementaci transportních metod pro IA simulaci (viz kapitola 4.3.1). Důležitým vylepšením je univerzálnost těchto metod vzhledem k použitým komunikačním protokolům. V původním řešení bylo nutné implementovat tyto metody samostatně pro každý protokol. Většina funkcionality byla totožná ve všech implementacích. Jediným rozdílem byla reprezentace

odpovědi, která závisí na konkrétním signálu v použitém protokolu. Informace o použitém datovém typu odpovědi a konverzní funkce z odpovědní položky payloadu jsou nově implementovány jako součást definice protokolu. Z pohledu rozhraní iniciátora mohou být použity univerzálně.

Volání transportních metod IA simulace vede na exekuci interní metody `DoRead`, nebo `DoWrite`. Hlavní zodpovědností těchto metod je příprava payloadu pro odeslání na připojené rozhraní. Instance payloadu je získána pomocí manažera payloadů, který je předán v konstruktoru rozhraní. Rozhraní iniciátora nastavuje adresu, operaci, velikost dat a v případě zápisové operace nastavuje i data. Payload s validními informacemi je odeslán do připojeného rozhraní cíle. Po vykonání operace je z payloadu vyčten stav odpovědi a v případě operace čtení i data.

Odeslání payloadu do připojeného rozhraní je možné díky TLM soketu iniciátora (viz kapitola 5.3.2). Třída `InitiatorInterface` dědí ze třídy `tlm::simple_initiator_socket`. To umožňuje přístup k metodám soketu přímo přes instanci rozhraní, což je vhodné například pro propojování komponent. Třída `InitiatorInterface` obsahuje pomocné metody `blocking`, `debug` a `transport`, které slouží jako zjednodušené obálky nad metodami TLM soketu. Instance TLM soketu vyžaduje určení šířky datové sběrnice a komunikačního protokolu. Obě hodnoty lze určit z parametrů rozhraní. Třída `InitiatorInterface` také obsahuje callback pro zpětnou komunikaci od připojeného rozhraní. Tento callback je použitý pouze pro CA simulaci a může být předefinován rozhraními pro konkrétní komunikační protokoly. Výchozí implementace považuje vyvolání tohoto callbacku za chybu.

Poslední částí třídy `InitiatorInterface` je sada pomocných metod pro přístup k payloadu, které jsou využity při IA i CA simulaci. Metoda `FixAddress` odstraňuje bity adresy, které přesahují velikost adresy. Metody `CopyDataToPayload` a `GetDataFromPayload` slouží pro konverzi dat mezi formátem payloadu a Codasip simulátoru. Tato konverze je popsána v kapitole 6.4.

Třída `TargetInterface` reprezentuje základní třídu pro všechna rozhraní cílů jednotlivých komunikačních protokolů. Dědí od třídy `Interface` a obsahuje celkem čtyři šablonové parametry. Jedná se o použitou třídu `InterfaceParams`, datový typ reprezentující připojenou paměť, datový typ interního rozhraní iniciátora a statickou třídu obsahující informace o právech ke čtení a zápisu. Třída `TargetInterface` obsahuje interní rozhraní iniciátora, které je nutné pro správné fungování simulátoru. Simulátor a debugger mohou tímto způsobem přistupovat přímo k rozhraní cíle. Z důvodu přítomnosti rozhraní iniciátora jsou ve třídě `TargetInterface` přítomny stejné metody pro IA simulaci jako ve třídě `InitiatorInterface`. Zde slouží pouze k přeposílání volání do instance interního rozhraní iniciátora.

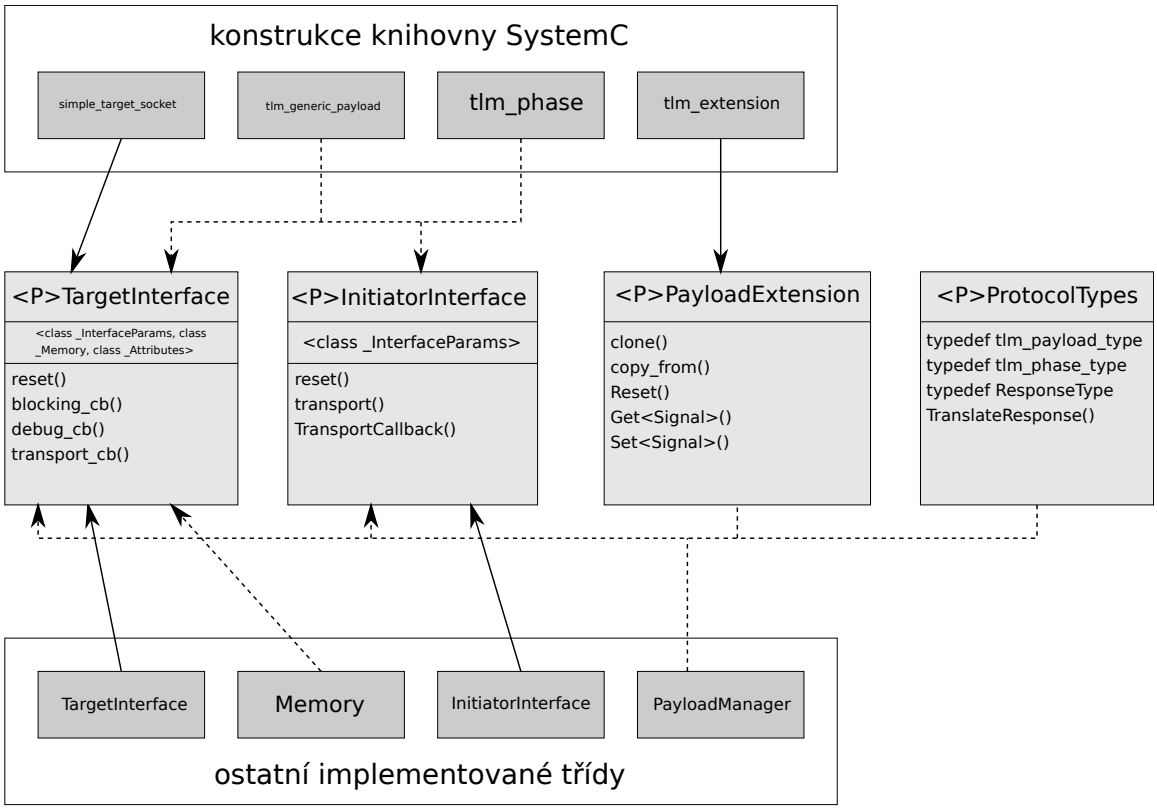
Další vlastností třídy `TargetInterface` je práce s latencí. Latence je parametrem paměti, nikoliv rozhraní. Přístup k latencím paměti je možný pomocí metod, které jsou definovány ve řídě `MemoryCore`. Codasip nástroje umožňují definovat latenci paměti zvlášť pro operace čtení a zvlášť operace zápisu. Každá z těchto definic latence může mít několik hodnot, které reprezentují možné zpoždění paměťové komponenty. Hodnoty jsou periodicky střídány, čímž vzniká simulace nestálé latence paměti. Rozhraní poskytuje metodu `GetNextLatency`, která vrací aktuální hodnotu latence pro danou operaci. Tato metoda je používána rozhraními jednotlivých komunikačních protokolů.

Poslední částí třídy `TargetInterface` je metoda `Check` pro kontrolu příchozích operací. Pokud kontrola selže je nastavena chybová odpověď payloadu. Kontrola ověřuje, zda rozhraní má oprávnění vykonat daný typ operace.

7.3 Implementace komponent pro komunikační protokoly

Pro každý podporovaný komunikační protokol je implementováno několik tříd, které jsou zobrazeny na obrázku 7.6.

- **Rozhraní iniciátora**: rozšiřuje třídu `InititatorInterface` o transportní metody volané z CA simulace a o neblokující volání TLM soketu.
- **Rozhraní cíle**: rozšiřuje třídu `TargetInterface` o neblokující volání TLM soketu.
- **Definice protokolu**: definuje protokol pro použití v TLM (viz kapitola 5.3.4) a obsahuje informace pro konverzi odpovědi v IA simulaci.
- **Rozšíření payloadu**: definuje rozšíření payloadu obsahující signály specifické pro komunikační protokol, které nelze reprezentovat položkami základního payloadu.



Obrázek 7.6: Diagram implementovaných konstrukcí specifických rozhraní. Plná čára značí dědičnost, přerušovaná čára značí použití. Značky <P> a <Signal> reprezentují název konkrétního komunikačního protokolu a názvy signálů v rozšíření payloadu.

Definice protokolů a rozšíření payloadů se nacházejí v následujících třídách:

- **AhbProtocolTypes**, **AhbPayloadExtension**
- **AxiProtocolTypes**, **AxiPayloadExtension**
- **CpbProtocolTypes**, **CpbPayloadExtension**
- **CpbLiteProtocolTypes**

Každá definice protokolu obsahuje použité datové typy a konverzní funkci pro odpověď transakce. Definované typy `tlm_payload_type` a `tlm_phase_type` jsou nutné pro správnou konfiguraci TLM soketů. Ostatní položky jsou využity třídou `InitiatorInterface` pro práci s odpovědí transakce v IA simulaci.

Rozšíření payloadů obsahují datové položky reprezentující určité signály sběrnice (viz kapitola 6.4) a metody pro práci s těmito položkami. Všechna rozšíření payloadů dědí od třídy `tlm::tlm_extension`. Sběrnice CPB-lite nedefinuje rozšíření payloadu, protože všechny její signály lze konvertovat na položky základního payloadu.

7.3.1 Obecné principy rozhraní iniciátora

Rozhraní iniciátorů pro jednotlivé protokoly jsou implementovány v následujících třídách:

- **AhbInitiatorInterface**
- **AxiInitiatorInterface**
- **CpbInitiatorInterface**
- **CpbLiteInitiatorInterface**

Tyto třídy dědí od bázové třídy `InitiatorInterface`. Základní princip fungování rozhraní iniciátora pro CA simulaci je podobný pro všechny typy protokolů. Rozdíly mezi protokoly jsou patrné z transportních metod a z interního fungování komunikace.

Jak bylo vysvětleno v kapitole 6.4, je každé volání transportní metody v rozhraní iniciátora ekvivalentní k jedné komunikační fázi TLM protokolu. Pokud je daná komunikační fáze vyvolávána iniciátorem, je rozhraní zodpovědné za získání payloadu, nastavení patřičných informací a odeslání payloadu připojenému rozhraní. Pokud rozhraní cíle poskytuje informace v rámci této fáze, je rozhraní iniciátora povinno tyto informace zpracovat a zapsat do patřičných zdrojů simulátoru. Tyto fáze musí být vždy kompletně zpracovány rozhraním cíle. Není nutné v těchto situacích používat další pomocné fáze pro dodatečný přenos informací.

Druhým typem jsou fáze vyvolávané rozhraním cíle. Tyto fáze představují pro implementaci komunikace největší problém. Kvůli nim bylo potřeba najít vhodný způsob synchronizace sekvenční a objektově řízené simulace (viz kapitola 7.1). Pomocí tohoto implementovaného principu má rozhraní iniciátora možnost počkat na danou fázi transakce a případně zjistit, že fáze nebude v hodinovém cyklu uskutečněna. V Případě, že daná fáze nenastane, musí rozhraní iniciátora nastavit předdefinované hodnoty signálů, aby byla transportní metoda korektně dokončena z pohledu procesoru. Některé fáze zahájené rozhraním cíle vyžadují, aby rozhraní iniciátora poskytlo zpět dodatečné informace, například zápisová data v protkolu AHB3-lite. Není však možné zajistit, že tyto informace bude mít

rozhraní iniciátora k dispozici při vykonání dané fáze, jelikož SystemC simulace nemá určeno pořadí aktivace jednotlivých procesů. Pokud tato situace může nastat v rámci dané fáze, rozhraní iniciátora odpovídá cíli návratovým kódem `tlm::TLM_ACCEPTED`. Tím indikuje poslání dodatečných informací zpět cíli v další pomocné fázi. V této práci musí být pomocné fáze posílány ve stejném hodinovém cyklu, jako fáze, které je vyvolaly. V tomto ohledu se implementace protokolů v této práci výrazně liší od implementace doporučeného TLM protokolu, kde mohou být pomocné fáze poslány v libovolném simulačním čase.

Každé specifické rozhraní iniciátora, které přijímá transakce od rozhraní cíle, musí definovat vlastní metodu `TransportCallback`. Typicky tato metoda uloží přijatý payload k dalšímu zpracování, které proběhne při zavolání příslušné transportní metody rozhraní. Pokud je tento payload dostupný již při vyvolání dané transportní metody, může být ihned zpracován. V opačném případě je rozhraní iniciátora společně s celým procesorem uspáno na jeden simulační delta cyklus. Kontrola přijetí payloadu je prováděna do dosažení maximálního počtu delta cyklů, kdy je fáze považována za neaktivovanou.

7.3.2 Obecné principy rozhraní cíle

Rozhraní cílů pro jednotlivé protokoly jsou implementovány v následujících třídách:

- **AhbTargetInterface**
- **AxiTargetInterface**
- **CpbTargetInterface**
- **CpbLiteTargetInterface**

Všechny tyto třídy dědí od bázové třídy `TargetInterface`. Každá z nich také dědí od třídy `tlm_utils::simple_target_socket`, která jí dává vlastnosti TLM soketu. Základní princip fungování rozhraní cíle pro CA simulaci je podobný pro všechny typy protokolů. Veřejné metody těchto tříd jsou totožné, protože všechny zpracovávají TLM komunikaci. Interní implementace tohoto zpracování se liší pro jednotlivé protokoly. Rozhraní cílů nemají definován vlastní simulační proces, který by řídil jejich chod. Existují pouze dva způsoby, jak ovlivnit chod rozhraní cíle.

Prvním způsobem je příchod transakce od připojeného rozhraní iniciátora. Typickým chováním při přijetí transakce je kontrola informací obsažených v payloadu a vyplnění požadovaných položek, které budou předány zpět iniciátorovi. Za předpokladu, že fáze obsahuje platnou paměťovou operaci, je od paměti zjištěna aktuální latence pro tuto operaci. Poté je paměťová operace naplánována na příslušný hodinový cyklus simulace. Plánovaní je realizováno pomocí PEQ fronty, která umožňuje uchovávat payloady i identifikace fází. Zároveň aktivuje událost simulačního jádra pro vyzvednutí těchto dat v konkrétním simulačním čase. Aktivace události simulačním jádrem vede na volání metody `QueueCallback`. Automatické zavolání této metody simulačním jádrem je druhým způsobem, jak ovlivnit chod rozhraní cíle. V rámci této metody se typicky posílají transakční fáze zahajované rozhraním cíle. Také se v této části vykonává operace v paměti za předpokladu, že rozhraní cíle má všechny potřebné informace k jejímu provedení. V případě, že rozhraní cíle potřebuje dodatečné informace od iniciátora, je toto předání informací realizováno pomocí další pomocné fáze zahájené iniciátorem. Pomocné fáze jsou zpracovávány stejnou metodou jako běžné fáze. Reakce na pomocné fáze jsou odlišné a specifické pro konkrétní protokoly. Za předpokladu, že pomocná fáze obsahovala data nutná k provedení paměťové operace, je

tato operace okamžitě vykonána. Při existenci pomocných fází je nutné kontrolovat nové typy chyby v komunikaci. Tyto chyby jsou vyvolány, pokud je přijata pomocná fáze, která není v dané situaci očekávána.

Každé rozhraní iniciátora i cíle obsahuje části specifického chování pro daný protokol, které rozšiřují nebo přesněji specifikují obecné chování popsané v této kapitole. Specifické části chování zde budou podrobněji rozebrány. Ostatní chování, které nebude specifikováno u konkrétního protokolu, je totožné s obecným chováním rozhraní. Transportní metody rozhraní iniciátora rozlišují mezi vstupními a výstupními parametry. Parametry předávány konstantní hodnotou jsou posílány do rozhraní cíle. Parametry předávány šablonovou referencí jsou zdroje simulátoru pro uložení přijatých informací. Jedinou výjimkou jsou parametry reprezentující data a datové masky, kterou jsou předávány pomocí šablonového typu, ale může se jednat o vstupní parametry.

7.3.3 Rozhraní pro protokol AHB3-lite

Rozhraní iniciátora protokolu AHB3-lite obsahuje tři transportní metody. Konverze hodnot a sémantika fází jsou definovány v kapitole 6.4.1.

```
// Metoda reprezentující fázi BEGIN_REQ
void transport_address(const uint8_t htrans,
    const uint8_t hwrite,
    const codasip_address_t addr,
    const uint8_t hsize,
    const uint8_t hprot,
    const uint8_t hmastlock,
    const uint8_t hburst);

// Metoda reprezentující fázi BEGIN_RESP
// se společným zdrojem a cílem dat
template<class HReady, class HResp, class HWRDATA>
void transport_data(HReady& hready, HResp& hresp,
    HWRDATA&& hrwdata);

// Metoda reprezentující fázi BEGIN_RESP
// s rozdílným zdrojem a cílem dat
template<class HReady, class HResp, class HRDATA, class HWDATA>
void transport_data(HReady& hready, HResp& hresp,
    HRDATA& hrdata, HWDATA&& hwdata);
```

V metodě `transport_address` je kromě běžného chování přítomna logika pro nastavení speciálního příznaku, který indikuje existenci nedokončené operace. Na základě tohoto speciálního příznaku je určena hodnota signálu `HREADYOUT` v případě neprovedené datové fáze operace. Příznak je zneplatněn při úspěšném přijetí datové fáze transakce v metodě `TransportCallback`.

Rozhraní cíle pro protokol AHB3-lite má některé specifické vlastnosti. Především se jedná o kontrolu validní operace. Validní operace vede k vykonání zápisu či čtení v paměti. Adresní fáze nevalidních operací jsou posílány do rozhraní cíle. Jejich datové fáze ale nejsou naplánovány. Rozhraní cíle může využít hodnot některých signálů sběrnice i v případě nevalidní operace.

Sběrnice AHB3-lite neumožňuje provádět více operací zároveň. Zpracování nové adresní fáze je proto podmíněno časem naplánované aktivace datové fáze předchozí operace. Nová adresní fáze je ignorována, pokud předchozí validní operace není dokončena. Ignorované adresní fáze nesmí být použity, protože specifikace sběrnice umožňuje měnit parametry operace mezi hodinovými cykly, ve kterých cíl zpracovává předchozí operaci.

Dalším specifickým chováním pro rozhraní protokolu AHB3-lite je zpracování chyb. Sběrnice AHB3-lite vyžaduje zpracování chyby ve dvou hodinových cyklech (viz kapitola 3.3.2). Při detekci chyby v rozhraní cíle je do PEQ fronty přidána jedna položka pro každý z obou cyklů. Tím je zajištěno, že se datová fáze transakce vykoná dvakrát v následujících dvou hodinových cyklech a žádná nová transakce nebude po dobu zpracování chyby zpracována. Rozhraní iniciátora reaguje na chybovou datovou fázi stejným způsobem jako na nechybovou, jelikož provádí pouze překlad hodnot signálů.

7.3.4 Rozhraní pro protokol CPB

Rozhraní iniciátora protokolu CPB obsahuje tři transportní metody. Konverze hodnot a sémantika fází jsou definovány v kapitole 6.4.2. Datový typ `T` označuje třídu `InterfaceParams` s konkrétními parametry sběrnice.

```
// Metoda reprezentující fázi BEGIN_REQ
template <class Ready>
void transport_address(Ready& aready,
    const uint8_t avalid,
    const uint8_t write,
    const codasip_address_t addr,
    typename T::param_type wdata = 0u,
    typename T::write_strobe_param_type wstrb = 0u);

// Metoda reprezentující fázi BEGIN_RESP pro operace zápisu
template<class Valid, class Resp>
void transport_data(Valid& valid, Resp& resp);

// Metoda reprezentující fázi BEGIN_RESP pro operace zápisu nebo čtení
template<class Valid, class Resp, class RData>
void transport_data(Valid& valid, Resp& resp, RData& rdata);
```

Na rozdíl od rozhraní iniciátora protokolu AHB3-lite umožňuje protokol CPB indikaci neúspěšného zpracování adresní fáze signálem `AREADY`. Hodnota tohoto signálu je vyžadována procesorem. Proto je nutné vyvolat adresní fázi i při neplatné operaci. Takové operace jsou rozhraním cíle zpracovány, ale nejsou pro ně naplánovány datové fáze. Obě metody pro transport datové fáze tohoto rozhraní fungují podobným způsobem a dokáží zpracovávat operace čtení i zápisu. Pouze metoda neobsahující parametr `rdata` neukládá signál obsahující přečtená data a pro operace čtení je tedy nevhodná.

Implementace rozhraní cíle protokolu CPB nepodporuje zpracování více operace zároveň. Podobně jako sběrnice AHB3-lite nemůže zpracovat další operaci dokud není dokončena operace předchozí. Na rozdíl od sběrnice AHB3-lite je tento přístup vlastností implementace a ne specifikace sběrnice. Zařízení typu slave na sběrnici CPB může umožňovat příjem dalších operací bez nutnosti dokončení aktuální operace. Tato funkcionalita nebyla v rámci této

práce implementována a nebyla podporována ani původní verzí rozhraní CPB v Codasip simulátoru.

Sběrnice CPB neimplementuje žádné pomocné fáze komunikace. Datová fáze sběrnice nevyžaduje žádné potvrzení nebo dodatečná data od rozhraní iniciátora.

7.3.5 Rozhraní pro protokol CPB-lite

Rozhraní iniciátora protokolu CPB-lite obsahuje tři transportní metody. Konverze hodnot a sémantika fází jsou definovány v kapitole 6.4.2. Datový typ `T` reprezentuje třídu `InterfaceParams` s konkrétními parametry sběrnice.

```
// Metoda reprezentující fázi BEGIN_REQ
void transport_address(const uint8_t avalid,
    const uint8_t write,
    const codasip_address_t addr,
    typename T::param_type wdata = 0u,
    typename T::write_strobe_param_type wstrb = 0u);

// Metoda reprezentující fázi BEGIN_RESP pro operace zápisu
template<class Resp>
void transport_data(Resp& resp);

// Metoda reprezentující fázi BEGIN_RESP pro operace zápisu nebo čtení
template<class Resp, class RData>
void transport_data(Resp& resp, RData& rdata);
```

Rozhraní iniciátora pro protokol CPB-lite neobsahuje speciální příznak nedokončené operace na rozdíl od protokolu CPB. Zařízení sběrnice CPB-lite mohou přijímat transakce v každém hodinovém cyklu. Příznak nedokončené operace by nebyl využit.

Zajímavostí rozhraní cíle tohoto protokolu je nepoužívání latence. Latence zařízení na této sběrnici je pevně definována standardem na jeden hodinový cyklus. Díky tomu není nutné potvrzování adresní ani datové fáze, protože obě strany komunikace jsou schopny přijmout tyto transakce v každém hodinovém cyklu. Není potřeba ani časové známky určující následující datovou fázi.

Díky vlastnostem sběrnice CPB-lite je tento protokol nejsnadněji implementovatelný ze všech podporovaných protokolů v této práci.

7.3.6 Rozhraní pro protokol AXI4-lite

Rozhraní iniciátora protokolu AXI4-lite obsahuje pět transportních metod. Konverze hodnot a sémantika fází jsou definovány v kapitole 6.4.3.

```
// Metoda reprezentující fázi BEGIN_REQ
template<class AReady>
void transport_read_address(const uint8_t avalid, AReady&& aready,
    const codasip_address_t aaddr, const uint8_t aprot);

// Metoda reprezentující fázi AXI_WRITE_ADDR
template<class AReady>
void transport_write_address(const uint8_t avalid, AReady&& aready,
    const codasip_address_t aaddr, const uint8_t aprot);

// Metoda reprezentující fázi BEGIN_RESP
template<class RValid, class RResp, class R>
void transport_read_data(RValid&& rvalid, const uint8_t rready,
    RResp&& rresp, R&& rdata);

// Metoda reprezentující fázi AXI_WRITE_DATA
template<class WReady, class W, class WStrobe>
void transport_write_data(const uint8_t wvalid, WReady&& wready,
    W& wdata, WStrobe& wstrb);

// Metoda reprezentující fázi AXI_WRITE_RESP_BEGIN
template<class BValid, class BResp>
void transport_write_response(BValid&& bvalid, const uint8_t bready,
    BResp&& bresp);
```

Rozhraní iniciátora protokolu AXI4-lite obsahuje více transportních metod než rozhraní ostatních podporovaných protokolů. Větší množství transportních metod je důsledkem přítomnosti dodatečných fází, které tento protokol popisuje. Jelikož sběrnice AXI4-lite obsahuje handshake signály pro každou komunikační fázi, iniciátor si nemusí uchovávat příznaky pro nedokončené operace. Všechny tři fáze zahajované iniciátorem mají podobnou implementaci. Na základě příslušného signálu typu `VALID` je určena validita operace. Validní i nevalidní operace musí být poslány do rozhraní cíle, protože procesor požaduje hodnotu signálu typu `READY` příslušného kanálu.

Zpracování fází, které jsou zahájeny rozhraním cíle, je v případě této sběrnice složitější než u ostatních podporovaných protokolů. V rámci sběrnice AXI4-lite má zařízení typu master možnost odmítnout příchozí fázi operace. Tato informace je předána do rozhraní cíle pomocí parametrů **`rready`** a **`bready`** příslušných transportních metod. V případě, že hodnota těchto signálů je logická nula, jsou veškeré informace v rámci přijaté fáze ignorovány. Rozhraní cíle je povinno stejnou fázi se stejnými informacemi poslat následující hodinové cykly, dokud nebude fáze korektně přijata iniciátorem. Za tímto účelem jsou payloady obsahující tyto informace interně ukládány rozhraním cíle. Odstraněny jsou až v případě, že rozhraní iniciátora potvrdí převzetí. Rozhraní iniciátora nemusí mít hodnoty zmíněných signálů k dispozici při zpracování dané komunikační fáze. Proto musí iniciátor vyvolat pomocnou fázi, ve které pošle cíli hodnotu příslušného signálu.

Protokol AXI4-lite definuje dvě komunikační fáze, které jsou zahájeny rozhraním cíle. V rozhraní iniciátora jsou implementována dvě nezávislá úložiště pro příchozí payloady, aby bylo možné od sebe odlišit informace získané z jednotlivých fází. Podobně je zdvojená i implementace rozhraní cíle. Fáze týkající se čtení a zápisu používají pouze stejnou PEQ frontu.

Pro operace zápisu je nutná implementace dodatečné logiky pro spojování informací z adresních a datových fází. Pro každou z těchto fází je vytvořená fronta payloadů. První aktivovaná fáze pro danou operaci zápisu je uložena do příslušné fronty. Druhá fáze vyzvedne payload první fáze z fronty a zpracuje oba payloady zároveň. Implementačně se jedná o kopii dat a datové masky z datového do adresního payloadu. Datový payload je následně uvolněn a s adresním payloadem jsou prováděny další standardní operace, především naplánování další fáze. Tyto speciální implementace se týkají pouze operací zápisu. Fáze pro operace čtení mají podobnou implementaci jako fáze předchozích protokolů.

Negativní vlastností implementace tohoto protokolu je nutnost dodržování určitého pořadí vykonávání fází ze strany procesoru. Je důležité, aby metody reprezentující fáze `AXI_WRITE_ADDR` a `AXI_WRITE_DATA` byly v každém cyklu volány až po metodě reprezentující fázi `AXI_WRITE_RESP_BEGIN`. Podobné pravidlo platí i pro metody reprezentující fáze `tlm::BEGIN_REQ` a `tlm::BEGIN_RESP`. Toto omezení je způsobeno implementací, nikoliv definicí protokolu. Podobně jako u implementace protokolu CPB je i zde omezen počet souběžně vykonávaných operací na jednu operaci čtení a jednu operaci zápisu. Rozhraní cíle neví, zda může danou operaci dokončit, protože dokončení operace závisí na hodnotách ze zmíněných fází. Proto tyto fáze musí proběhnout nejdříve, aby rozhraní cíle vědělo, zda jsou operace dokončeny. Na základě této informace může cíl přijmout další operace poslané v příslušných fázích. Toto omezení je striktně kontrolováno na straně rozhraní cíle.

Kapitola 8

Integrace, testování, použití

Implementované řešení bylo integrováno do Codasip simulátoru. Bylo náležitě otestováno pomocí jednotkových testů i na praktických příkladech. Toto řešení bylo následně optimalizováno a byla změřena jeho rychlost v porovnání s původní implementací Codasip simulátoru.

8.1 Integrace

Pod pojmem integrace zde rozumíme úpravy Codasip nástrojů, které jsou nutné pro správné fungování nových komponent v Codasip simulátoru. Prvním krokem byla úprava zdrojových souborů simulátoru, které byly vygenerovány původními nástroji. Především se jednalo o úpravy vyžadované knihovnou SystemC a o změnu instancí pamětí a rozhraní z původní implementace na novou.

V generované části paměti bylo nutné upravit definici interního paměťového úložiště. Dále byly nahrazeny původní definice rozhraní za nově implementované komponenty. Aby bylo možné předat rozhraním cílů potřebné parametry pro jejich vytvoření, bylo nutné přeposlat tyto parametry přes konstruktor paměti. Generovaná část paměti se nově nachází v prostoru jmen `codasip::systemc`, čímž se odlišuje od původní implementace umístěné v prostoru jmen `codasip::resources`.

Generovaný simulátor procesorového jádra také musel být upraven. Především byly nahrazeny původní instance rozhraní a pamětí nově implementovanými komponentami. Tyto komponenty již nevyužívají metodu `clock_cycle` pro posun simulačního času a její volání proto musí být odstraněno. Dále byl v rámci simulátoru instanciován manažer payloadů, který je používán všemi přítomnými rozhraními. Třída reprezentující simulátor nově dědí od třídy `sc_core::sc_module` z knihovny SystemC. To umožňuje zaregistrovat metodu `Run` jako SystemC proces řídící hlavní simulační smyčku procesoru. Definice metody `Run` vypadá následovně:

```
void Sim::Run()
{
    // Zahájení simulace
    int rc = SIM_OK;
    do {
        // Vykonání cyklu procesoru
        rc = Sim::ClockCycle();
        // Volání simulačního jádra knihovny SystemC
        wait(m_ClockPeriod);
    } while (rc == SIM_OK);
    // Uložení návratového kódu simulace
    SetReturnCode(rc);
}
```

Dalším krokem integrace je úprava Codasip nástrojů, které generují simulátor. Cílem je, aby generátor simulátoru dokázal z modelu automaticky vytvořit zdrojové kódy s popsanými úpravami. Kromě těchto úprav je potřeba několik dalších změn, aby bylo zajištěno správné generování a překlad výsledného simulátoru.

Důležité jsou úpravy třídy `SimulatorPrivateInterface`, která definuje základní metody, které musí vygenerovaný simulátor implementovat. Nachází se zde především deklarace metody `Run`. Dále je nutné přidat proměnnou pro předání návratového kódu simulátoru. Tato informace byla původně předávána návratovou hodnotou metody `Run`. Tento způsob už není podporován, kvůli registraci metody `Run` do SystemC simulace. Tato metoda sloužila jako hlavní startovní bod simulátoru. Nová implementace spouští CA simulaci pomocí metody `sc_core::sc_start()` z knihovny SystemC.

Důležitou součástí integrace je správné volání transportních metod rozhraní. Tyto metody jsou volány z generovaného simulátoru procesoru. Jejich použití definuje tvůrce modelu naplánováním jejich vykonání v konkrétních situacích. Aby nemusely být prováděny změny v modelech procesorů, byly ve všech rozhraních iniciátorů implementovány transportní metody se stejnou deklarací jako transportní metody původního simulátoru. Tyto dodatečné transportní metody se liší od metod popsaných v kapitole 7.3 pouze svým jménem a přidaným parametrem určujícím fázi transakce. V rámci integrace se tyto nové metody staly primárním způsobem přístupu k rozhraní iniciátora. Deklarace nových transportních metod pro protokol AHB3-lite vypadá následovně:

```
void transport(const Phase phase,
    const uint8_t htrans,
    const uint8_t hwrite,
    const codasip_address_t haddr,
    const uint8_t hsize,
    const uint8_t hprot,
    const uint8_t hmastlock,
    const uint8_t hburst);

template<class HReady, class HResp, class HWRDATA>
void transport(const Phase phase, HReady& hready, HResp& hresp,
    HWRDATA&& hwrdata);

template<class HReady, class HResp, class HRDATA, class HWDATA>
void transport(const Phase phase, HReady& hready, HResp& hresp,
    HRDATA& hrdata, HWDATA&& hwdata);
```

Metody pro ostatní protokoly jsou implementovány obdobným způsobem. Pouze pro protokol AXI4-lite bylo nutné sloučit některé fáze do jedné transportní metody kvůli kolizi datových typů parametrů. Implementace nových metod se skládá z kontroly parametru fáze a zavolání transportních metod definovaných v kapitole 7.3.

Pro překlad vygenerovaného simulátoru je využito nástroje *Cmake* [23]. Jedinou provedenou změnou v souborech nástroje Cmake bylo přidání knihovny SystemC pro překlad a linkování zdrojových souborů simulátoru. Nástroj Cmake poskytuje automatický způsob nalezení a přidání podporovaných balíků pomocí následujícího kódu:

```
set(SYSTEMC_DIR "<cesta_k_systemc_instalaci>")
find_package(SystemCLanguage CONFIG REQUIRED HINTS "${SYSTEMC_DIR}")
target_link_libraries(isimulator PRIVATE SystemC::systemc)
# isimulator je název targetu reprezentující binární soubor simulátoru.
```

8.2 Testování

Pro ověření funkčnosti nově vytvořených komponent byla implementována sada jednotkových testů. Tyto testy ověřují samostatně většinu implementace komponent. Kromě použití jednotkových testů bylo výsledné integrované řešení testováno na reálných procesorových modelech.

Většina praktických testů využila model procesoru *Codasip urisc*. Jedná se o jeden z jednodušších a ukázkových procesorů společnosti Codasip, na kterém je testována velká část funkčnosti všech Codasip nástrojů. Model *Codasip urisc* je používán v mnoha verzích. Jsou k dispozici verze implementující přímé propojení procesoru s pamětí pomocí komunikačních protokolů. Tyto verze byly upravovány a konfigurovány pro testování různých situací. Použitím těchto modelů byla ověřena funkčnost nového simulátoru na kombinaci všech podporovaných komunikačních protokolů, všech typů endianity dat a několika hodnot latence paměti. Simulátory byly testovány na několika aplikacích vytvořených pomocí Codasip nástrojů. Hlavní aplikace pro testování byly *Bitcnt*, *Dhrystone* a *Coremark* [40, 18].

Vytvořené jednotkové testy odhalily řadu chyb v prvních verzích implementace. Tyto chyby byly postupně odstraňovány. Ve finální verzi implementace neodhalují jednotkové

testy žádné chyby. Pro vytvoření těchto testů byl použit systém *Google Test* [20], který je využíván pro testování Codasip nástrojů. Všechny nově implementované třídy byly pokryty těmito testy. Testovány byly všechny implementované komunikační protokoly, IA i CA simulace a rozhraní s různými parametry. Tyto parametry zahrnují několik podporovaných datových typů, oba druhy endianity a různé šířky adresy.

Propojení systému Google Test a knihovny SystemC představovalo komplikaci při vytváření jednotkových testů pro komponenty použité v CA simulaci. Testy pro rozhraní iniciátorů a cílů bylo obtížné implementovat, protože využívají simulačního jádra knihovny SystemC. SystemC simulace nemůže být v rámci jednoho programu spuštěna opakovaně. Z tohoto důvodu je pro každý test vytvořen vlastní zdrojový soubor, který je přeložen do samostatného programu. Tímto způsobem je možné v rámci testu spustit SystemC simulaci pro konkrétní komponenty.

Další komplikací bylo zajištění kontroly správnosti běhu pro komponenty v SystemC simulaci. Tento problém byl vyřešen definicí očekávaných testovacích transakcí. Testovací transakce jsou rozděleny podle jejich typu, který určuje mezi kterými komponentami mají být posílány a v jakém směru. Pro testování každé komponenty jsou vytvořeny zjednodušené modely ostatních připojených komponent. Úkolem těchto modelů je přijímat a odesílat předem definované transakce a kontrolovat korektní hodnoty těchto transakcí. Obdobným způsobem byly implementovány testy pro komplexní kontrolu celého paměťového subsystému.

8.3 Optimalizace a výkonnost

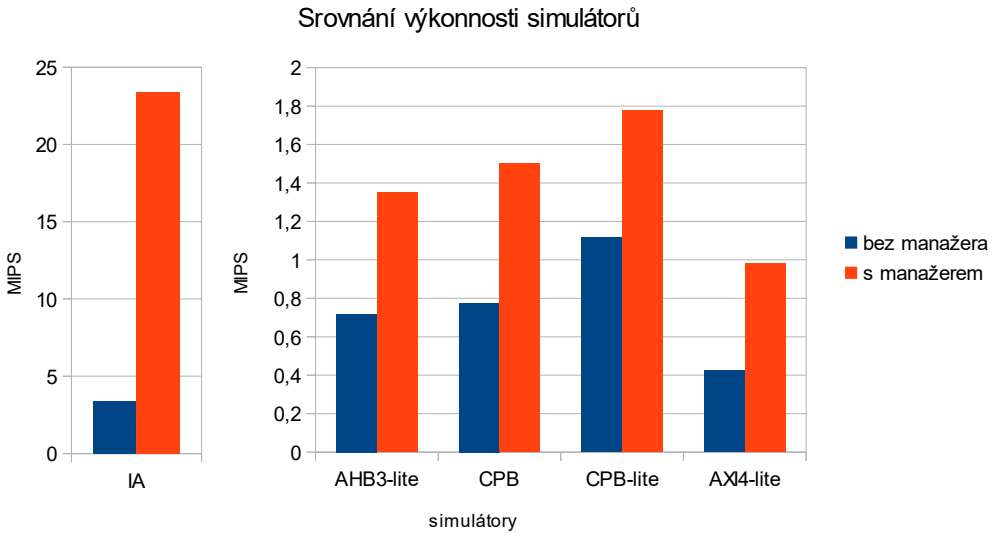
Během vývoje paměťového subsystému byla do implementace postupně zakomponována celá řada optimalizací a drobných vylepšení.

Nejdůležitější optimalizace se týkají práce s payloady. Alokace payloadů je jedním z největších problémů při řešení výkonnosti. První implementační pokusy alokovaly instance payloadů vždy při jejich použití a uvolňovaly paměť ihned po ukončení jejich platnosti. Během běhu jednoduché simulace bylo alokováno a dealokováno tisíce payloadů. Tím byla simulace značně zpomalena. Řešením tohoto problému byla implementace manažera payloadů, který zastřešuje správu paměti pro všechny payloady simulace. Payload manažer byl navržen takovým způsobem, aby instance payloadů nebyly odstraněny po jejich využití, ale byly určeny pro opětovné použití. Touto optimalizací je omezena režie při práci s pamětí, což vede ke zrychlení simulace. Při vývoji práce byla do manažera payloadů přidána funkcionalita pro režii paměti dodatečných součástí payloadů. Mezi tyto součásti patří pole dat, datová maska a rozšíření payloadu pro jednotlivé protokoly. Implementace manažera payloadů je popsána v kapitole 7.2.1.

Práce s payloady byla dále optimalizována. Pomocí metod `acquire` a `release` dostupných v TLM payloadu je možné využít instanci payloadu se všemi obsaženými informacemi i po provedení transportní metody, ve které byl payload použit. Tento systém lze nejvíce využít při neblokujících transportních metodách. Jednu instanci payloadu lze využít zároveň ve více, někdy i všech, fázích dané operace. Tímto řešením se vyhneme případné alokaci dodatečných payloadů a kopírování podstatných dat z payloadů, které by byly za normálních okolností odstraněny. Použitá optimalizace zajišťuje, že payload, který je dostupný ve fázi provádění paměťové operace, obsahuje již při doručení všechny potřebné informace z předchozích fází. Objekt payloadu v jednotlivých fázích je díky této optimalizaci v mnoha případech totožný. Transakční informace nutné pro vykonání paměťové operace jsou do

tohoto objektu postupně přidávány a jsou persistentní mezi jednotlivými fázemi operace. Tato optimalizace se může potenciálně stát i součástí definic TLM protokolů.

Srovnání výkonnosti simulace s použitím manažera payloadů a s klasickou alokací payload objektů je zobrazeno v grafu 8.1.



Obrázek 8.1: Graf měření výkonnosti v závislosti na použití manažera payloadů.

Mezi další implementované optimalizace patří především práce s daty v operacích čtení a zápisu. Počáteční verze subsystému prováděly převod mezi celočíselnou hodnotou a polem bajtů po jednotlivých bajtech. Každý bajt dat byl samostatně získán z celočíselné hodnoty a zkopírován do příslušné položky pole. V případě opačné konverze byl každý bajt pole vhodným způsobem započítán do výsledné hodnoty. Ve většině situací se lze vyhnout přístupu k jednotlivým bajtům dat. Lze použít metodu `std::memcpy` pro jednorázové kopírování celého bloku dat. Pro převod endianity se využívá metoda `Byteswap` z Codasip nástrojů.

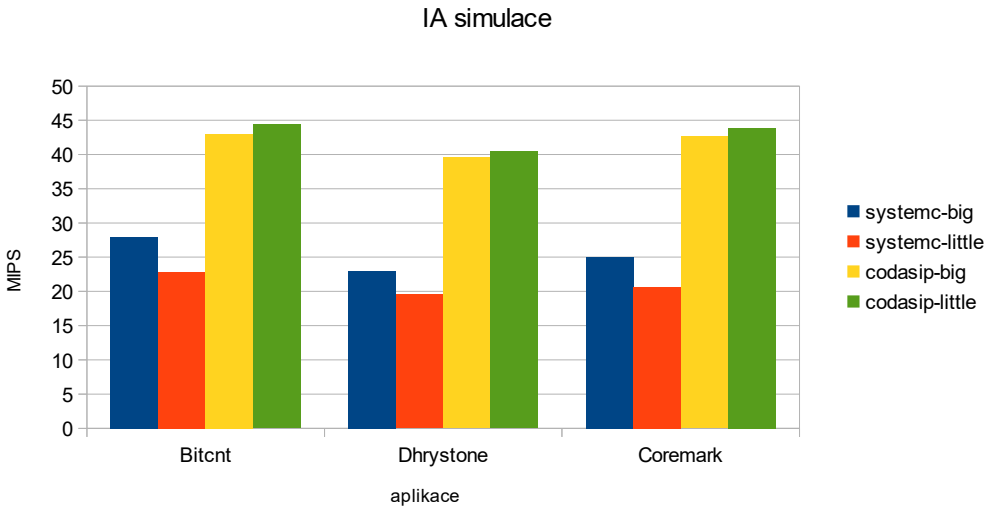
Významnou optimalizací bylo také omezení využití datové masky. Ve velkém množství případů jsou data operace využita celá a není tedy důvod kontrolovat platnost jednotlivých bajtů pomocí masky. Tato skutečnost platí především pro IA simulaci, kde datová maska určovala vždy všechny bajty jako platné. Dalším příkladem jsou operace čtení pro většinu podporovaných sběrnic, kde podle specifikace nelze datovou masku určit. Z důvodu přítomnosti těchto situací byla implementována optimalizace datové masky. V konečném řešení je datová maska použita pouze v CA simulaci a pouze v případech, kdy daná sběrnice podporuje operace s daty, které nemusí mít platné všechny bajty přenášené po sběrnici. V případech, kdy není potřeba datové masky, je tato položka v přenášeném payloadu nevyplněna. Podle specifikace TLM jsou v takovém případě využity všechny bajty slova.

IA simulace byla optimalizována díky nepoužití simulačního jádra knihovny SystemC. Protože veškeré paměťové operace v IA simulaci jsou okamžité, nezáleží při jejich vykonávání na aktuálním simulačním čase. Díky této skutečnosti by v simulaci existoval pouze jediný SystemC proces, proces chodu procesoru, který by posouval simulační čas. V takovém případě je možné simulační jádro knihovny SystemC ignorovat bez negativních důsledků.

Tato optimalizace přispěla ke zvýšení výkonu simulátoru typu IA. Použití vylepšené konverze dat i odstranění datové masky v nepotřebných případech také výrazně zvýšilo výkon simulátoru, protože práce s daty a s datovou maskou je prováděna při každé operaci.

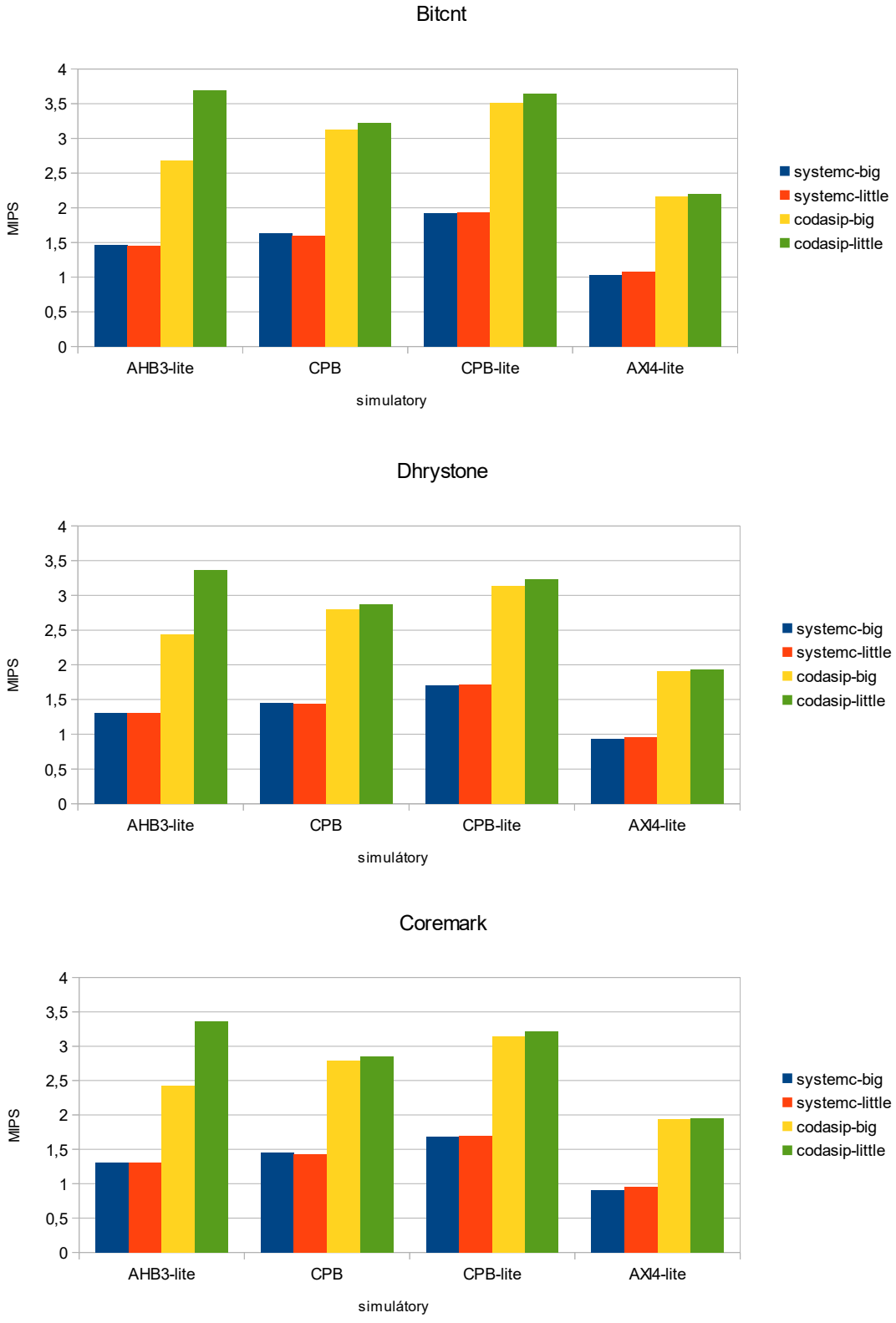
8.3.1 Měření výkonnosti

Měření výkonnosti simulátoru bylo provedeno na všech komunikačních protokolech a obou typech endianity. Využito bylo výkonnostních testů *Dhrystone* a *Coremark* a ukázkové aplikace *Bitcnt*. Výkonnost simulátoru využívajícího knihovnu SystemC a původního Codasip simulátoru se výrazně liší. Původní implementace je výkonnostně lepší než nová, jak lze vidět na grafech 8.3 a 8.2. Simulátory označené `systemc` reprezentující simulátor s novou implementací paměťového subsystému. Simulátory označené `codasip` reprezentují původní implementaci simulátoru. Označení `little` a `big` určuje typ endianity rozhraní.



Obrázek 8.2: Grafy měření výkonnosti simulátorů typu IA na aplikacích *Bitcnt*, *Dhrystone* a *Coremark*.

Podrobné informace o měření výkonu jsou umístěny v příloze C.



Obrázek 8.3: Grafy měření výkonnosti simulátorů typu CA na aplikacích *Bitcnt*, *Dhrystone* a *Coremark*.

Pokles výkonnosti má několik hlavních důvodů. Většina z nich byla detekována pomocí profilování výsledného simulátoru.

- Neoptimální implementace manažera payloadů: v IA simulaci je manažer payloadů z pohledu výkonu nejkritičtějším místem. Práce s payloady v IA simulaci by mohla být dále optimalizována například existencí pouze jednoho payload objektu pro celý simulátor. Při každém použití by se měnila pouze velikost datového pole. Tato optimalizace je implementovatelná díky okamžitému vykonávání operací v IA simulaci. Payload manažer pro CA simulaci by také bylo možné dále optimalizovat, například pomocí vylepšení kategorizace payload objektů.

- Kopírování dat: přes snahy optimalizace konverze dat jsou data stále kopírována při každé operaci na obou stranách komunikace. Tomuto kopírování by bylo možné se v některých případech vyhnout. Například ladící transportní metody nebo komunikační metody v IA simulaci by mohly využít přístupu DMI, který je implementován v knihovně SystemC.

- Využití SystemC simulace: využití knihovny SystemC pro simulaci paměťového subsystému také snižuje výkon výsledného řešení. Z důvodu zachované simulace procesoru musí být přítomno i původní simulační jádro implementované v Codasip nástrojích. Simulátor provádí podobnou simulaci jako v původní implementaci a zároveň provádí další simulaci jiného typu, což zhoršuje výkon simulátoru. Potenciálním řešením by bylo implementovat ostatní součásti Codasip simulátoru za pomoci knihovny SystemC a využít při této implementaci simulační jádro této knihovny. Původní implementace simulačního jádra by mohla být odstraněna a simulace by byla řízena pouze jedním centrálním způsobem.

- Synchronizace simulačních metod: dalším důvodem ztráty výkonnosti je kombinace dvou simulačních metod popsaných v kapitole 7.1. Zvolené řešení je univerzálně použitelné pro zpracování všech typů komunikace. Často ale využívá volání simulačního jádra knihovny SystemC, které celkovou simulaci zpomaluje. Řešení zmíněné v předchozím bodu by částečně vyřešilo i tento problém, protože by nebyla nutná synchronizace mezi dvěma typy simulace.

- Větší množství komunikačních fází: důležitým faktorem výkonnosti je existence více komunikačních fází, než bylo v komunikaci mezi původními komponentami. Nové pomocné fáze, vytvořené pro bližší přiblížení k doporučenému TLM protokolu, přidávají komplexnost většině operací a tím snižují výkon simulátoru.

Snížená výkonnost simulátoru byla očekávanou vlastností výsledného řešení. Zrychlení simulátoru nebylo primárním cílem této práce. Výkonnost by mohla být zvýšena vytvořením kompletně nového komunikačního protokolu. Tím by byla ztracena vazba na doporučený TLM protokol. Takové řešení by mělo za následek potenciální zrychlení simulace, ale také větší nekompatibilitu s TLM specifikací.

8.4 Rozšíření

Implementace této práce, přestože je plně funkční, může být v budoucnu doplněna o dodatečnou funkcionalitu a případně upravena pro specifické potřeby Codasip nástrojů. Inspirací pro možná rozšíření je původní implementace Codasip simulátoru. V rámci této implementace existuje velké množství dodatečné funkcionality, která v rámci této práce nebyla převzata.

Jedním z příkladů možného rozšíření je implementace obálek nad novými komponentami. Tyto obálky mohou plnit různou funkci. Většinou rozšiřují původní komponentu o dodatečné kontroly nebo nově prováděné operace. Jedním z příkladů těchto obálek v Codasip Simulátoru je tzv. *dumper*. Účelem této obálky je evidence transakcí, které byly přijaty či odeslány danou komponentou. Dalším příkladem je tzv. *protocol checker*. Tato obálka slouží k dodatečné kontrole dodržování daného komunikačního protokolu. Využití má například pro sběrnici AHB3-lite. Zde mohou být kontrolovány korektní inkrementace adresy v blokových operacích, stálé kontrolní signály transakce v čekajícím stavu, apod. Další možnou obálkou je implementace kontroly zarovnání adresy a dat. Tato kontrola byla původně umístěna přímo v komponentách rozhraní. Z důvodu optimalizace výkonu by bylo vhodnější implementovat ji jako volitelné rozšíření.

Použití obálek by bylo možné podmínit konfigurací při generování simulátoru. Uživatel by si mohl zvolit, které obálky použít na jednotlivé komponenty. Lze vytvořit například specializovaný simulátor pro maximální kontrolu správnosti, pro jednodušší ladění nebo pro maximální rychlost simulace. Podobný princip je použit v rámci komponent původního Codasip simulátoru a mohl by být potenciálně využit i pro nové komponenty.

Dalším možným rozšířením je podpora některých aktuálně nepodporovaných částí paměťového subsystému. Jedno z omezení aktuální implementace spočívá v nutnosti osmibitové nejmenší adresovatelné jednotky na všech komponentách. Odlišná hodnota není často používaná, ale je vhodná pro některé speciální situace. Implementace odlišné nejmenší adresovatelné jednotky než je velikost bajtu na hostitelském systému je problematická. Data by musela být konvertována do pole bajtů, které používá Knihovna SystemC. Tato konverze musí správným způsobem přepočítat rozdíly na úrovni bitů.

Další nepodporovanou funkcionalitou je možnost vykonávat více operací zároveň na rozhraní cíle. Některé cílové komponenty mohou mít schopnost přijímat další operace při nedokončeném zpracování aktuální operace. Účelem může být paralelní zpracování transakcí nebo ukládání dat do mezipaměti. Tato funkcionalita nebyla podporována ani v původních komponentách Codasip simulátoru.

Codasip nástroje poskytují vlastní implementaci datových typů pro vyjádření hodnot o různých bitových velikostech. Simulátor může být rozšířen o podporu datových typů definovaných v knihovně SystemC (viz kapitola 5.1.2). Možnosti těchto dvou implementací jsou podobné. Využití datových typů z knihovny SystemC by přispělo ke zjednodušení zdrojového kódu simulátoru. Proprietální implementace těchto datových typů v Codasip nástrojích by mohla být odstraněna.

8.4.1 Tvorba nových komponent

Primárním cílem této práce bylo standardizovat základní komponenty paměťového subsystému simulátoru pomocí knihovny SystemC a principu TLM. Tento cíl byl splněn, ale simulátor stále obsahuje komponenty, které by mohly být přepracovány podobným způsobem.

Ne všechny tyto komponenty jsou součástí paměťového subsystému. Codasip simulátor poskytuje vlastní implementaci komponent pro interní použití v simulaci procesoru. Jedná se o registry, signály, porty a další podobné komponenty. Pro některé z těchto komponent existují implementace v knihovně SystemC, které je možné integrovat do simulátoru. Komponenty, které nejsou v knihovně SystemC implementovány, lze vytvořit vhodnou kombinací jiných konstrukcí. Implementace a použití těchto komponent v Codasip simulátoru mimo paměťový subsystém může být dalším rozšířením této práce.

Codasip simulátor implementuje komponenty paměťového subsystému, které v této práci nejsou podporovány. Jedná se o propojovací spoje, vyrovnávací paměti, arbitry a další komponenty, které mohou být umístěny do cesty mezi procesor a cílovou periferii. Tyto komponenty je možné implementovat za pomoci knihovny SystemC a principu TLM. K implementaci nové komponenty je nutné znát pouze její specifikaci a definici komunikačního protokolu, který má být použit pro její připojení k ostatním komponentám.

Zde je nutné upozornit na skutečnost, že implementovaná rozhraní iniciátorů a cílů nemusí být vždy vhodným vstupním či výstupním bodem komponenty. Rozhraní iniciátorů jsou primárně určena pro komunikaci mezi procesorem a paměťovým subsystémem. Rozhraní cílů jsou primárně určena pro komunikaci mezi pamětí a zbytkem paměťového subsystému. Tato rozhraní provádí zpracování daného protokolu a jsou buď počátečními nebo koncovými komponentami při TLM komunikaci. Pro implementaci komponent zapojených mezi koncové body je ve většině případů vhodné použít základní TLM sokety. Tyto komponenty nezpracovávají transakce stejně jako výchozí a cílové zařízení. Jejich hlavní funkcionalitou je především přeposílání transakcí mezi vstupním a výstupním bodem komponenty a provádění vhodných změn transakčních informací. Mezi další časté funkcionality patří zpožďování transakcí nebo předčasné ukončování transakcí před zpracováním cílovou komponentou.

Tvorba dalších komponent paměťového subsystému je potenciálním rozšířením této implementace. Zjednodušení procesu vytváření komponent bylo jedním z cílů práce. Nové komponenty by měly být vytvářeny snadněji než v původní implementaci Codasip simulátoru. Především se jedná o zjednodušení pro uživatele Codasip nástrojů. Implementace nových komponent pro simulaci byla velice obtížná a v některých případech i nemožná. Například nebylo možné zapojit Codasip simulátor typu CA jako komponentu do větších SystemC simulací. Tvorba komponent byla omezena proprietální implementací a návrhem komunikačního protokolu a rozhraní. Nyní může nové komponenty implementovat každý uživatel se znalostí knihovny SystemC, principu TLM a změn komunikačního protokolu oproti doporučenému TLM protokolu. Není nutné používat proprietální rozhraní nebo znát implementaci komunikujících prvků.

Kapitola 9

Závěr

Paměťový subsystém pro Codasip simulátor byl navržen a implementován s pomocí knihovny SystemC a principu TLM. Řešení bylo otestováno a integrováno do Codasip simulátoru. Všechny primární cíle práce byly tímto splněny.

Nový paměťový subsystém je přehlednější a umožňuje lepší znovupoužitelnost díky konstrukcím z knihovny SystemC. Především použití komunikačních protokolů, payloadů a soketů z části TLM knihovny SystemC přispívá ke sjednocení způsobu komunikace a jeho jednoduchému opětovnému využívání ve všech částech paměťového subsystému. Přechod na SystemC také umožňuje pohodlnější a jednodušší implementaci nových komponent jak v paměťovém subsystému tak i v jiných částech simulátoru. Podporovány jsou všechny požadované sběrnice AHB3-lite AXI4-lite, CPB a CPB-lite. Standardizované komunikační protokoly založené na TLM umožňují připojit simulátor do rozsáhlejších simulačních systémů bez nutnosti znalosti implementace rozhraní. Tyto protokoly lze využít i samostatně bez Codasip nástrojů.

Výsledné řešení je oproti původní implementaci pomalejší. Je možné implementaci dále optimalizovat s cílem dosažení lepší výkonnosti simulace. Některé z komponent, které byly implementovány v původním paměťovém subsystému, nejsou v této práci podporovány. Jedná se například o vyrovnávací paměti nebo arbitry. Je možné tyto komponenty implementovat v rámci rozšíření práce, a to jednodušším způsobem za použití knihovny SystemC a nově definovaných TLM protokolů. Podobným způsobem lze implementovat i další komponenty a rozšíření. Tuto práci lze použít jako základ pro další rozšiřování a modifikace Codasip simulátoru.

Literatura

[1] Open On-Chip Debugger. [online], 2020.
URL http://openocd.org/

[2] Aldec: Riviera-PRO. [online], 2020.
URL
https://www.aldec.com/en/products/functional_verification/riviera-pro

[3] ARM Limited: AMBA 3 AHB-Lite Protocol Specification. [online], 2006.
URL https://silver.arm.com/download/download.tm?pv=1085658

[4] ARM Limited: AMBA AXI and ACE Protocol Specification. [online], 2019.
URL
https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf

[5] Aynsley, J.: Complete TLM-2.0 AT Example, including Initiators, Interconnect and Targets. Leden 2009.
URL https://www.doulos.com/knowhow/systemc/tlm2/at_example/

[6] Black, D. C.; Donovan, J.: SystemC: From the Ground Up. [online], 2004.
URL https:
//pdfs.semanticscholar.org/5684/347243eb12364f863171c6a8c7b07c47df12.pdf

[7] Black, D. C.; Donovan, J.; Bunton, B.; aj.: *SystemC: From the ground up*. 2010, ISBN 978-0-387-69957-8.

[8] Cadence Design Systems: Xcelium Parallel Logic Simulator. [online], 2020.
URL https:
//www.cadence.com/en_US/home/tools/system-design-and-verification/
simulation-and-testbench-verification/xcelium-parallel-simulator.html

[9] Codasip: Codasip Studio Technical Reference Manual. [online], 2020.
URL https://support.codasip.com/download/?filename=
Codasip+Studio+Technical+Reference+Manual.pdf&resource=
%2Fprivate%2Fdownload%2Fgroups%2Fdoc%2Fmanuals%2Fdeveloper

[10] Codasip: Codasip Studio User Guide. [online], 2020.
URL https://support.codasip.com/download/?filename=
Codasip+Studio+User+Guide.pdf&resource=
%2Fprivate%2Fdownload%2Fgroups%2Fdoc%2Fguides%2Fdeveloper

[11] Codasip: Codasip web page. [online], 2020.
URL https://codasip.com/

[12] Daněk, M.: Programovatelná hradlová pole – FPGA. *Automa – časopis pro automatizační techniku*, 2006.
URL https://automa.cz/cz/casopis-clanky/programovatelna-hradlova-pole-fpga-2006_02_30930_672/

[13] FIT VUT v Brně: Paměti. [online], 2019.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FINP-IT%2Flectures%2Finp2019_12pameti.pdf&cid=10322

[14] FIT VUT v Brně: Sběrnice a periferní zařízení. [online], 2019.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FINP-IT%2Flectures%2Finp2019_14bus_per.pdf&cid=10322

[15] Foster, H.: Part 10: The 2018 Wilson Research Group Functional Verification Study - IC/ASIC Language and Library Adoption Trends . [online], Únor 2019.
URL https://blogs.mentor.com/verificationhorizons/blog/2019/02/14/part-10-the-2018-wilson-research-group-functional-verification-study/

[16] Free Software Foundation: GNU Binutils. [online], 2017.
URL https://www.gnu.org/software/binutils/

[17] Free Software Foundation: objdump (GNU Binary Utilities). [online], 2019.
URL https://sourceware.org/binutils/docs/binutils/objdump.html

[18] Gal-On, S.; Levy, M.: Exploring CoreMark—A benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[19] Ghenassia, F.; aj.: *Transaction-level modeling with SystemC*. 2005, ISBN 978-0-387-26232-1.

[20] Google: Google Test. [online], 2020.
URL https://github.com/google/googletest

[21] IEEE: IEEE Standard for Standard SystemC Language Reference Manual. [online], Leden 2012.
URL https://ieeexplore.ieee.org/servlet/opac?punumber=6134617

[22] IEEE: IEEE Standard for Universal Verification Methodology Language Reference Manual. [online], Květen 2017.
URL https://ieeexplore.ieee.org/servlet/opac?punumber=7932210

[23] Kitware: Cmake web page. [online], 2020.
URL https://cmake.org/

[24] Lavagno, L.; Markov, I. L.; Martin, G.; aj.: *Electronic Design Automation for IC System Design, Verification, and Testing*. CRC Press, 2017.

[25] LLDB Team: The LLDB Debugger. [online], 2020.
URL https://lldb.llvm.org/

[26] LLVM Foundation: The LLVM Compiler Infrastructure. [online], 2019.
URL https://llvm.org/

[27] Mentor, a Siemens Business: ModelSim. [online].
URL https://www.mentor.com/products/fv/modelsim/

[28] Mentor, a Siemens Business: Questa Advanced Simulator. [online].
URL https://www.mentor.com/products/fv/questa/

[29] Michl, K.: Ladění software v Codasip Studiu pomocí JTAG rozhraní simulovaném v RTL simulátoru. 2016.

[30] PCI-SIG: PCI. [online], 2020.
URL https://pcisig.com/

[31] Pech, J.: Programovatelná logika – část 5: Jazyky pro popis logických obvodů. *DPS Elektronika od A do Z*, 2015.
URL https://www.dps-az.cz/vyvoj/id:22197/programovatelna-logika-cast-5-jazyky-pro-popis-logickych-obvodu

[32] Přikryl, Z.: Implementace obecného zpětného assembleru. 2007.

[33] Rath, D.: Open On-Chip Debugger. 2008.

[34] SATA-IO: SATA. [online], 2020.
URL https://sata-io.org/

[35] Shiva, S. G.: Computer hardware description languages—A tutorial. *Proceedings of the IEEE*, ročník 67, č. 12, 1979: s. 1605–1615.

[36] Stroustrup, B.: *The C++ programming language fourth edition*. 2013, ISBN 978-0-321-56384-2.

[37] Synopsys: VCS. [online], 2020.
URL https://www.synopsys.com/verification/simulation/vcs.html

[38] Thornton Scott: What's the difference between Von-Neumann and Harvard architectures? 2018.
URL https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/

[39] USB Implementers Forum: USB. [online], 2019.
URL https://www.usb.org/

[40] Weiss, A. R.: Dhrystone benchmark. *History, Analysis,„Scores "and Recommendations, White Paper, ECL/LLC*, 2002.

[41] Xilinx: Xilinx. [online], 2020.
URL https://www.xilinx.com/

[42] XJTAG: What is JTAG and how can I make use of it? [online], 2020.
URL https://www.xjtag.com/about-jtag/what-is-jtag/

[43] Šimková, M.: Hardwarově akcelerovaná funkční verifikace. 2010.

Příloha A

Program využívající TLM

```
#include <cstring>
#include <iostream>
#include <systemc>
#include <tlm>
#include <simple_initiator_socket.h>
#include <simple_target_socket.h>
#define ADDR_MAX 64

class Memory : public sc_core::sc_module { public:
    uint8_t* m_Data;
    tlm_utils::simple_target_socket<Memory> m_Socket;

    Memory(const sc_core::sc_module_name name)
        : sc_core::sc_module(name),
          m_Data(new uint8_t [ADDR_MAX]) {
        m_Socket.register_b_transport(this, b_transport_callback);
    }

    ~Memory() { delete m_Data; }

    void b_transport_callback
        (tlm::tlm_generic_payload& payload, sc_core::sc_time& delay) {
        if ((payload.get_address()+payload.get_data_length()) > ADDR_MAX) {
            payload.set_response_status(tlm::TLM_ADDRESS_ERROR_RESPONSE);
            return;
        }
        if (payload.is_read()) {
            memcpy(payload.get_data_ptr(), m_Data + payload.get_address(),
                payload.get_data_length());
        }
        if (payload.is_write()) {
            memcpy(m_Data + payload.get_address(), payload.get_data_ptr(),
                payload.get_data_length());
        }
        payload.set_response_status(tlm::TLM_OK_RESPONSE);
```

```
    }
};

class Processor : public sc_core::sc_module { public:
    tlm_utils::simple_initiator_socket<Processor> m_Socket;

    Processor(const sc_core::sc_module_name name, Memory* mem)
        : sc_core::sc_module(name) {
        m_Socket.bind(mem->m_Socket);
    }

    void Operation(bool write, uint64_t address,
                   unsigned int size, uint8_t* data) {
        tlm::tlm_generic_payload payload;
        payload.reset();
        if (write) payload.set_write(); else payload.set_read();
        payload.set_address(address);
        payload.set_data_length(size);
        payload.set_data_ptr(data);
        payload.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
        sc_core::sc_time delay = sc_core::SC_ZERO_TIME;
        m_Socket->b_transport(payload, delay);
        if (!payload.is_response_ok()) {
            std::cout << "Operation failed" << std::endl;
        }
    }
};

int main () {
    Memory* mem = new Memory("memory_instance");
    Processor* proc = new Processor("processor_instance", mem);
    uint8_t* data;

    data = new uint8_t [4] {0x0, 0x1, 0x2, 0x3};
    proc->Operation(true, 0x10, 4, data);
    proc->Operation(false, 0x10, 4, data);
    // Print read data
    delete data;
    data = new uint8_t [2] {0x44, 0x55};
    proc->Operation(true, 0x5, 2, data);
    proc->Operation(false, 0x5, 2, data);
    // Print read data
    delete data;

    delete proc; delete mem;
    return 0;
}
```

Příloha B

Obsah přiloženého paměťového média

Přiložené paměťové medium obsahuje všechny nově vytvořené zdrojové kódy pro tuto práci a ukázky použití výsledného simulátoru. Simulátor není možné úspěšně přeložit bez potřebných zdrojových souborů a knihoven, které jsou součástí Codasip nástrojů. Přiložené simulátory nelze spustit bez licence pro použití Codasip nástrojů.

Výsledné simulátory je možné spustit použitím binárního souboru konkrétního simulátoru s parametrem simulované aplikace ve formátu *xexe*. Všechny binární soubory jsou přeloženy pomocí nástroje *MinGW* a jsou určeny pro operační systém *Windows*.

Podrobnější informace o spouštění přiložených simulátorů jsou obsaženy v souboru *README.txt*.

- **README.txt**: soubor obsahující obsah paměťového média a další důležité informace o použití programu.
- **Thesis.pdf**: soubor obsahující technickou zprávu ve finální podobě.
- **Codasip.lic**: soubor obsahující dočasnou licenci pro používání Codasip nástrojů.
- **applications**: adresář obsahující ukázkové aplikace pro spuštění simulátoru.
- **documentation**: adresář obsahující zdrojový kód technické zprávy v jazyce Latex.
- **simulators**: adresář obsahující přeložené binární soubory simulátorů.
- **sources**: adresář obsahující zdrojové kódy všech implementovaných součástí této práce.
- **unittests**: adresář obsahující jednotkové testy k implementovaným komponentám.

Příloha C

Měření výkonnosti

Testování a měření výkonnosti bylo provedeno na operačním systému *Windows 10* typu *64-bit* s procesorem *Intel(R) Core(TM) i7-4790 CPU 3.60GHz* a pamětí RAM o velikosti 32 GB. Všechny simulátory byly přeloženy pomocí překladače *gcc* a nástroje *MinGW*. Byla použita konfigurace `release` a optimalizační parametr `-O3`. Počty instrukcí jednotlivých testovacích aplikací byly změřeny pomocí počtu vykonaných cyklů v IA simulaci těchto aplikací. Měření času vykonávání simulace bylo uskutečněno pomocí příkazu `Measure-Command` z programu *Powershell*. Všechna měření byla vykonána několikrát a výsledné hodnoty byly zprůměrovány. Doba vykonávání aplikace v simulátoru je dostatečně dlouhá na to, aby bylo možné zanedbat režii spojenou s nahráváním aplikace a inicializací simulátoru. Pro IA simulace bylo využito aplikací s větším počtem iterací za účelem lepších výsledků měření.

Instrukce	little	big
bitcnt IA (50 000)	1 101 750 076	1 101 750 076
bitcnt CA (5 000)	110 175 076	110 175 076
dhrystone IA (1 000 000)	1 253 113 878	1 253 113 878
dhrystone CA (100 000)	125 410 805	125 410 803
coremark IA (1 000)	1 443 641 227	1 453 787 214
coremark CA (100)	144 455 176	145 468 417

Tabulka C.1: Počty vykonaných instrukcí pro jednotlivé aplikace s různým počtem iterací a s různou endianitou.

IA	systemc-big	systemc-little	codasip-big	codasip-little
Bitcnt	39,43	48,2	25,61	24,82
Dhrystone	54,65	64,17	31,67	30,98
Coremark	58,15	70,15	34,1	32,89

Tabulka C.2: Průměrná doba trvání běhů jednotlivých simulátorů typu IA na různých aplikacích v sekundách.

Bitcnt	systemc-big	systemc-little	codasip-big	codasip-little
AHB3-lite	75,5	76,13	41,12	29,8
CPB	67,79	68,87	35,21	34,13
CPB-lite	57,44	57,12	31,35	30,2
AXI4-lite	106,67	101,97	50,86	50,19

Tabulka C.3: Průměrná doba trvání běhů jednotlivých simulátorů typu CA na aplikaci Bitcnt v sekundách.

Dhrystone	systemc-big	systemc-little	codasip-big	codasip-little
AHB3-lite	96,12	95,91	51,51	37,3
CPB	86,44	87,46	44,73	43,64
CPB-lite	73,55	73,13	40,11	38,77
AXI4-lite	135,48	130,57	65,81	64,94

Tabulka C.4: Průměrná doba trvání běhů jednotlivých simulátorů typu CA na aplikaci Dhrystone v sekundách.

Coremark	systemc-big	systemc-little	codasip-big	codasip-little
AHB3-lite	111,69	111,09	59,91	42,95
CPB	100,63	101,37	52,09	50,64
CPB-lite	86,34	85,12	46,33	44,88
AXI4-lite	159,7	151,96	75,28	74,0

Tabulka C.5: Průměrná doba trvání běhů jednotlivých simulátorů typu CA na aplikaci Coremark v sekundách.