
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**
**DEPARTMENT OF COMPUTER SYSTEMS**


# IMPLEMENTACE ROZHRANÍ 10GB ETHERNETU PRO ARRIA 10 SOC

**IMPLEMENTATION OF 10 GB ETHERNET INTERFACE FOR ARRIA 10 SOC**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **DAVID NOVÁK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. JAN KOŘENEK, Ph.D.**
**SUPERVISOR**



**BRNO 2018**

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačových systémů Akademický rok 2017/2018

# Zadání bakalářské práce

Řešitel: **Novák David**
Obor: Informační technologie
Téma: **Implementace rozhraní 10Gb Ethernetu pro Arria 10 SoC**
**Implementation of 10 Gb Ethernet Interface for Arria 10 SoC**
Kategorie: Návrh číslicových systémů

Pokyny:

1. Seznamte se s OS Linux, systémem DPDK a technologií Arria 10 SoC FPGA.
2. Navrhněte hardwarovou architekturu MAC vrstvy Ethernetu pro rychlost 10 Gb/s s ohledem na technologii FPGA Arria 10 SoC.
3. Proveďte implementaci MAC vrstvy s ohledem na syntézu do uvedeného FPGA.
4. Ověřte funkčnost a vlastnosti vytvořené implementace.
5. Navrhněte a implementujte DPDK rozhraní pro FPGA Arria 10 SoC.
6. Změřte vlastnosti vytvořené implementace DPDK rozhraní.
7. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího rozšíření.

Literatura:

- Dle pokynů vedoucího.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kořenek Jan, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
*vedoucí ústavu*

# Abstrakt

Tato práce se zabývá návrhem, implementací a testováním 10 Gb Ethernet rozhraní pro čip Arria 10 SoC (kombinace FPGA a ARM Cortex-A9). Je zde popsána podoba rozhraní, jeho součástí a komunikace mezi nimi. Hlavní pozornost je věnována MAC vrstvě, která byla v rámci práce navržena a implementována. Druhým aspektem práce je problém zvyšujících se nároků systémů pro zpracování paketů na výkon CPU. Při rychlostech 10 Gb/s a vyšších již výkon běžných procesorů nepostačuje a je nutné hledat alternativní řešení – konkrétně akcelerace některých úkonů v FPGA a využití nových způsobů práce s pakety. Součástí práce je proto popis DPDK (knihovny pro rychlé zpracování paketů) a implementace DPDK rozhraní pro vytvořený modul MAC.



# Abstract

This thesis addresses design, implementation and testing of 10 Gb Ethernet interface for chip Arria 10 SoC (combination of FPGA and ARM Cortex-A9). Composition of the interface, its parts and communication between them is described with main focus being on MAC layer, which was designed and implemented in the course of this work. Secondary aspect of this thesis is increasing CPU performance demands for processing of packets and problems it brings. The performance of common CPUs is seriously lacking with network speeds over 10 Gb/s and alternative solutions has to be considered – namely acceleration of some tasks using FPGA and utilization of new ways of packet processing. Therefore, the description of DPDK (library for fast packet processing) as well as implementation of DPDK interface for newly created MAC module, are part of this thesis.



# Klíčová slova

Ethernet, MAC, DPDK, Zásobník síťových protokolů OS Linux, Schránky, FPGA, Arria 10, VHDL, výkon síťových rozhraní

# Keywords

Ethernet, MAC, DPDK, Linux Network Stack, Network Sockets, FPGA, Arria 10, VHDL, performance of network interfaces

# Citace

NOVÁK, David. *Implementace rozhraní 10Gb Ethernetu pro Arria 10 SoC*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Kořenek, Ph.D.

# Implementace rozhraní 10Gb Ethernetu pro Aria 10 SoC

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kořenka, Ph.D. Další informace a konzultace mi poskytli Ing. Vlastimil Košař, Ing. Tomáš Fukač a Bc. Michal Orsák. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Novák
16. května 2018

## Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Janu Kořenkovi, Ph.D. za trpělivost a rady při psaní práce. Dále bych chtěl poděkovat Ing. Vlastimilu Košařovi za dlouhé hodiny technických konzultací a rad při návrhu a implementaci. Poslední poděkování patří Bc. Michalu Orsákovi za ochotné vysvětlení funkcionality jeho DMA řadiče a rady při implementaci DPDK rozhraní.

# Obsah


**1 Úvod** **2**

**2 Síťové aplikace pro OS Linux a DPDK** **4**
2.1 Síťový zásobník v OS Linux . . . . . 4
2.2 Standardní síťové rozhraní - schránky . . . . . 6
2.3 DPDK . . . . . 7
2.3.1 EAL . . . . . 7
2.3.2 Ring Manager . . . . . 8
2.3.3 Memory Pool Manager . . . . . 8
2.3.4 Network Packet Buffer Management . . . . . 9
2.3.5 Ovladače síťových zařízení . . . . . 11

**3 HW platforma** **12**
3.1 Arria 10 SoC . . . . . 12
3.2 DMA . . . . . 13

**4 Návrh 10 Gb Ethernet rozhraní** **14**
4.1 Popis 10 Gb Ethernet rozhraní . . . . . 15
4.2 Návrh MAC . . . . . 17

**5 Implementace** **21**
5.1 Popis použitých rozhraní . . . . . 21
5.2 MAC vrstva . . . . . 24
5.3 DPDK rozhraní . . . . . 26

**6 Testování a výsledky** **27**
6.1 VHDL testbench . . . . . 27
6.2 Funkcionální verifikace . . . . . 30
6.3 Ověření implementace na cílové platformě . . . . . 32
6.4 Výsledky . . . . . 34

**7 Závěr** **36**

**Literatura** **37**

# Kapitola 1

# Úvod

V posledních letech probíhal výrazný vývoj v oblasti síťových technologií. Narůstala zejména přenosová rychlost a také požadavky na zpracování a analýzu síťového provozu. 10 Gb Ethernet byl poprvé definován v roce 2002, zatímco v roce 2010 již byly k dispozici Ethernet zařízení s rychlostí 100 Gb/s a v době psaní práce dosahuje rychlost až 400 Gb/s.

Se zvyšováním rychlosti síťové komunikace rostou také požadavky na výkon nutný ke zpracování přenášených dat. Běžné CPU nejsou schopny zpracovat všechny příchozí pakety a tedy dochází ke ztrátě dat. Například pro 10 Gb Ethernet, definovaný normou IEEE 802.3[7], je maximální počet paketů na jedné lince (minimální velikosti 84 bajtů) za sekundu 14,8 milionů a čas na zpracování jednoho paketu je tedy 67,2 ns. Tento čas odpovídá přibližně 201 taktům na 3 GHz CPU a je naprosto nedostatečný pro jednoduché přijetí, natož analýzu paketu[2].

Protože Linux není navržen pro rychlé zpracování paketů, ale pro běh aplikací (velké množství podporované funkcionality, potřeba skládat síťové toky, atd.), není možné dosáhnout potřebného výkonu. Existují proto projekty, které vytváří jednodušší síťový protokolový zásobník od nuly. Jde například o DPDK (Data Plane Development Kit), které je specificky zaměřeno na vysoce výkonné síťové aplikace běžící kompletně v uživatelském prostoru[3].

I přes všechna zlepšení a využití nákladných více-procesorových systémů je ovšem výkon nedostatečný a problém zpracování a analýzy velkého množství paketů je obvykle nutné řešit využitím aplikačně-specifických obvodů (ASIC) nebo FPGA[1] čipů. Než ale mohou být síťová data takto zpracována, musí být přijata dle pravidel Ethernet protokolu. Toto zajišťuje Ethernet MAC, který byl v rámci této práce implementován. Po zpracování (typicky filtrace) jsou data dále odeslána do sítě nebo předána k dalšímu zpracování aplikaci běžící na běžném CPU.

Cílem této práce je navrhnout a implementovat 10 GbE[2] rozhraní včetně příslušných DPDK ovladačů pro využití v FPGA, konkrétně čipu Intel Arria 10. Práce se dále zabývá rozdíly (zejména výkonovými) mezi tradičním rozhraním schránek (Sockets), které je k dispozici v moderních operačních systémech, a rozhraním DPDK. Popsána je využitá platforma, její možnosti a také samotný návrh, způsob jeho implementace a verifikace. Výsledky této práce jsou využívány například v rámci projektu SProbe a nově vyvinuté 10 Gb platformy ARMINDA.

[1] Field Programmable Gate Array – programovatelné hradlové pole
[2] 10 Gigabit Ethernet

Práce je rozdělena na tři hlavní části – teoretickou část, popis návrhu a implementace a průběh testování spolu se shrnutím výsledků. Teoretická část se skládá z druhé a třetí kapitoly, které popisují síťový zásobník OS Linux, knihovnu DPDK a použitou platformu Arria 10 SoC. Následuje samotný návrh modulu MAC a popis jeho součástí. V páté kapitole jsou popsána jednotlivá rozhraní Arria 10 SoC, které byla použita v rámci 10 Gb Ethernet rozhraní. Dále se kapitola zabývá některými aspekty samotné implementace a také problémy, které bylo třeba vyřešit. Poslední kapitola pojednává o možnostech testování a verifikace VHDL komponent a jejich využití pro testování MAC.

# Kapitola 2

# Síťové aplikace pro OS Linux a DPDK

Síťové aplikace a síťová architektura se obvykle popisuje ve vztahu k síťovému modelu OSI (Open Systems Interconnection). Ten popisuje[11] sedm logických vrstev:

- Fyzická vrstva – specifikuje způsob fyzické komunikace
- Linková vrstva – zajišťuje komunikaci mezi dvěma sousedními zařízeními
- Síťová vrstva – řídí směrování paketů v síti a adresaci hostů
- Transportní vrstva – zajišťuje přenos dat mezi koncovými uzly a zaručuje požadovanou spolehlivost
- Relační vrstva – popisuje relační spojení a jeho synchronizaci mezi dvěma systémy
- Prezentační vrstva – stará se o formát a strukturu vyměňovaných dat
- Aplikační vrstva – poskytuje síťové služby uživatelským aplikacím

## 2.1 Síťový zásobník v OS Linux

Síťový zásobník (také síťový protokolový zásobník) je důležitou součástí OS Linux. Z pohledu síťového modelu OSI pracuje na druhé, třetí a čtvrté vrstvě (dále L2, L3, L4). L2 (linková vrstva) je reprezentována ovladači síťových zařízení (zejména Ethernet NIC[1]). V rámci L3 (síťová vrstva) jsou implementovány především protokoly IPv4 a IPv6. L4 (transportní vrstva) nabízí protokoly TCP a UDP. Vrstvy L5 a vyšší jsou zpracovávány výhradně v rámci uživatelských aplikací, fyzická vrstva je obsluhována v HW. Můžeme tedy říci, že skrze síťový zásobník putují pakety mezi uživatelským prostorem a ovladačem konkrétního síťového zařízení[15].

Ovladače síťových zařízení konfigurují a řídí síťový HW, poskytují informace o schopnostech a stavu zařízení, statistiky provozu a uživatelské rozhraní pro ovládání zařízení (využíváno programy `ip`, `ifconfig` či `ethtool`). Význačným nastavením z pohledu analýzy síťového provozu je tzv. promiskuitní režim, ve kterém nejsou zahazovány pakety, které jsou určené pro jiné hosty. Zahození může proběhnout v síťovém zásobníku OS i přímo v HW v případě neshodující se cílové MAC adresy.

[1] Network Interface Card

### Přijímání a odesílání paketů

Základem odesílání a příjmu paketů je systém přerušení (IRQ[2]). Každý příchozí paket má svůj deskriptor v RX[3] frontě síťového zařízení. Ten obsahuje adresu paměti, kam má být paket přenesen DMA řadičem. Po úspěšném dokončení přenosu je vygenerováno přerušení, které je obslouženo jádrem. V rámci této obsluhy je paket zkopírován do bufferu jádra (konkrétně struktury `sk_buff`) a následně prochází zásobníkem, kde dochází k jeho zpracování a filtraci[15]. Každý paket prochází směrovacím subsystémem, který určuje, zda má být paket poslán dále a případně na které rozhraní. Dále je vyhodnocen v subsystému `netfilter`, který je možné nastavovat například skrze známé `iptables`.

Paket může být ovlivněn několika dalšími subsystémy (například `IPSec` a multicast) a může být fragmentován, pokud jeho velikost přesahuje MTU[4]. Odesílání paketu začíná v L4 schránce (TCP nebo UDP) a probíhá analogicky s tím rozdílem, že přerušení signalizující možnost odeslání dalšího paketu je vyvoláno po dokončení DMA přenosu do TX fronty v síťovém zařízení.

Z předchozího popisu vyplývá, že síťový subsystém v OS Linux je poměrně komplikovaný, robustní a se širokou funkcionalitou. Tento přístup je vhodný pro běžné použití a umožňuje poměrně jednoduchou tvorbu pokročilých síťových klientských aplikací. Zároveň ale vede k výkonnostním problémům způsobených velkým množstvím vykonaných operací, přepínáním kontextu mezi jádrem a uživatelským procesem, vícenásobným kopírováním a vyvoláváním přerušení pro každý odeslaný a přijatý paket. Poslední problém byl vyřešen zavedením NAPI.

### NAPI

NAPI[5] přináší možnost práce síťových zařízení v tzv. polling režimu[5] namísto generování přerušení pro každý příchozí paket. Při vysoké zátěži je jádro zahlcováno velkým množstvím přerušení, které nepřináší žádnou novou informaci (jádro ví, že jsou k dispozici nové pakety, protože ještě nestihlo zpracovat ty, o kterých bylo informováno minulými přerušeními).

Po vygenerování prvního přerušení jsou tedy další zakázána a namísto toho se po krátké době jádro ovladače zeptá, kolik je k dispozici paketů a přesune je všechny zároveň. Poté může být generování přerušení opět povoleno. Ovladače s podporou NAPI (v současné době skoro všechny) také podporují zahazování paketů při přetížení přímo v HW – bez zásahu (a tedy přidané režie) jádra.

Využití NAPI zvyšuje výkon při vysoké zátěži sítě, ale přináší potencionální zvýšení latence[5]. Neřeší ovšem ostatní výkonnostní problémy síťového subsystému, a proto vzniklo několik projektů snažících se o implementaci jednoduššího síťového subsystému v uživatelském prostoru. V druhé části této kapitoly se budu věnovat jednomu z nich – DPDK.

[2] Interrupt Request - žádost o přerušení
[3] příjem, z angl. reception
[4] Maximum Transmission Unit - Maximální přenosová jednotka; standardní MTU pro Ethernet je 1500B
[5] New API, v Linuxu od verze 2.6

## 2.2 Standardní síťové rozhraní - schránky

Schránky (sockets) poskytují rozhraní mezi uživatelskou aplikací a síťovým subsystémem v jádře OS. Poprvé byly implementovány v OS BSD[6] v roce 1983 a následně převzaty a implementovány ostatními systémy. S evolučními úpravami se staly součástí standardu POSIX a obvykle jsou nazývány Berkley sockets nebo POSIX[7] sockets. Je nutné je odlišovat od Netlink sockets, které v OS Linux poskytují rozhraní pro přenos různých informací souvisejících se síťovým subsystémem. Netlink využívá například balíček nástrojů `iproute2`, který slouží pro konfiguraci síťových rozhraní, směrovacích tabulek, tunelů a získávání statistik.

Nejčastější použití schránek je k odesílání a přijímání paketů na L4 (transportní vrstvě). K dispozici jsou protokolové rodiny AF_INET a AF_INET6 pro protokoly IPv4 a IPv6 vždy v kombinaci s typem schránky SOCK_STREAM pro TCP nebo SOCK_DGRAM pro UDP. Detailní popis rozhraní je možné nalézt například v manuálových stránkách tcp(7) a udp(7).

Aplikace pro analýzu síťového provozu potřebují přistupovat i k nižším vrstvám – síťové a linkové. Přístup k L3 je řešen tzv. raw schránkami (typ SOCK_RAW), které umožňují přístup k paketům protokolů IP, ICMP a IGMP. Přístup k L2 běžné rozhraní schránek neumožňuje. V OS Linux byla k tomuto účelu přidána nová protokolová rodina AF_PACKET poskytující dva typy schránek:

- SOCK_RAW – poskytuje pakety včetně L2 hlavičky
- SOCK_DGRAM – L2 hlavička je odstraněna (příjem) nebo automaticky generována (odesílání)

Schránky na nižších vrstvách může z bezpečnostních důvodů vytvářet pouze privilegovaný proces (umožňují aplikaci přístup ke všem paketům).

[6] Berkley Software Distribution; OS Unixového typu

[7] Portable Operating System Interface; rodina IEEE standardů definující jednotné API s cílem zachovat kompatibilitu různých OS Unixového typu

## 2.3 DPDK

DPDK (Data Plane Development Kit) je skupina knihoven a ovladačů pro rychlé zpracování paketů[3] (s důrazem na minimální počet cyklů CPU). Původně bylo vyvíjeno společností Intel se zaměřením na OS Linux a platformu x86. DPDK je vydáváno pod BSD licencí a v současnosti je správcem projektu Linux Foundation (dříve Intel). Postupně byla doplněna podpora pro architekturu ARM a později IBM POWER. Část DPDK je také k dispozici pro OS FreeBSD. Je nutné poznamenat, že DPDK není úplný síťový zásobník a neposkytuje většinu funkcí, které nabízí OS Linux (například L3 přeposílání, IPsec nebo firewall). Pokud aplikace takovou funkcionalitu vyžaduje, musí ji sama implementovat.

Centrální filozofií DPDK je minimalizace režie při zpracování paketů a využití optimalizací, které jsou nabízeny danou platformou. Například:

- alokace všech prostředků před spuštěním aplikace a vlastní správa paměti
- využití dotazovacího režimu (tzv. polling)
- kopírování dat probíhá pouze jednou – ze zařízení rovnou do uživatelského prostoru
- zpracování více jádry (několik RX/TX front)
- vyhrazení CPU jádra pro jeden proces/vlákno – nedochází k přepínání kontextu
- běh v uživatelském prostoru – neprobíhá přepínání kontextu mezi jádrem a aplikací
- huge pages[8] – vyšší úspěšnost (hitrate) vyhledávání adresy v TLB[9] (nižší režie překladu VA[10] na PA[11])

Tímto je oproti běžnému zpracování paketů odstraněna značná část režie, ale zároveň je omezena flexibilita (většina funkcionality síťového zásobníku OS Linux není dostupná a musí být v případě potřeby implementována v aplikaci).

### 2.3.1 EAL

EAL (Environment Abstraction Layer; `librte_eal`) poskytuje společné rozhraní pro knihovny DPDK a vytváření DPDK aplikací nezávisle na použité platformě[3]. EAL poskytuje například tyto služby:

- načítání a spuštění DPDK
- správa paměti
- identifikace CPU a kontrola kompatibility (aplikace může například vyžadovat přítomnost určitého rozšíření instrukční sady)
- obsluha přerušení (například při odpojení přenosového média) a časovačů (možnost nastavit funkce s konkrétním časem spuštění)
- atomické/zamykací operace

[8] Obrovské stránky paměti; obvykle 2MB, ale mohou být i větší
[9] Translation Lookaside Buffer
[10] Virtual Address - virtuální adresa
[11] Physical Address - fyzická adresa

- přístup ke sběrnicím
- funkce pro sledování a ladění aplikace

Inicializace EAL probíhá před spuštěním samotné DPDK aplikace (ve funkci `main` je volána funkce `rte_eal_init`) v tzv. master vláknu. Ostatní vlákna jsou označena jako slave a čekají, než jim bude přiřazena funkce. Jednotlivá vlákna je možné přiřadit konkrétním jádrům CPU pomocí EAL parametru `--lcores`. Ideální stav je identický počet jader a vláken, čímž dosáhneme eliminace režie spojené s přepínáním kontextu. Pozitivní vliv na výkon má ale i svázání několika vláken na jedno jádro. Důvodem je využívání stejné L1 a L2 cache stejnými vlákny a tedy zvýšení pravděpodobnosti, že cache již obsahuje správná data.

Při spouštění DPDK aplikace jsou zadávány dvě skupiny parametrů – EAL a aplikační. Tyto od sebe oddělujeme pomocí "--". Na OS Linux využívá EAL knihovny pthread (pro vytváření vláken) a libc[12].

### 2.3.2 Ring Manager

Knihovna `librte_ring` poskytuje bez-zámkový (díky použití atomických operací) kruhový buffer typu FIFO pevné velikosti. Ring je primárně určen pro asynchronní komunikaci mezi jednotlivými procesy/vlákny DPDK aplikace a podporuje současně více producentů i více konzumentů. Typické použití může být jedno vlákno přijímající veškeré pakety ze síťového zařízení a ukládající je do x front, kde každá náleží jednomu procesu. Rozdělením zátěže na více jader můžeme dosáhnout vyšší maximální propustnosti aplikace[3].

### 2.3.3 Memory Pool Manager

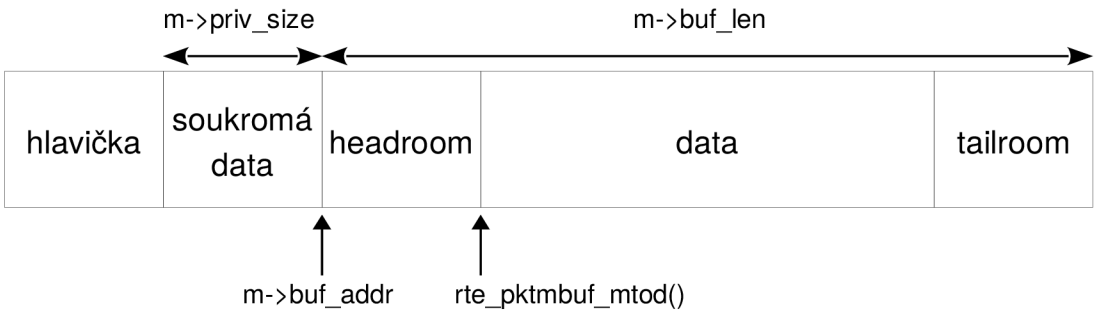
Knihovna `librte_mempool` zprostředkovává alokaci objektů pevné velikosti a počtu a je optimalizována pro vysokou výkonnost. Alokace těchto objektů probíhá při inicializaci aplikace a k jejich ukládání je použit Ring. Aplikace pak může objekty získávat a zase vracet. Mempool[13] nabízí i pokročilou funkcionalitu k další výkonové optimalizaci – jmenovitě samostatná cache objektů pro každé jádro CPU a možnost zarovnat objekty na určité adresy k zajištění jejich rovnoměrného rozložení přes dostupné kanály hlavní paměti. Knihovna je také připravena pro provoz v NUMA[14] systémech, kde je žádoucí, aby každé jádro využívalo svou lokální paměť[3].

Cache objektů zvyšuje výkon omezením počtu CAS (compare-and-set) operací, které jsou vyžadovány při přístupu do `mempool`. Namísto přístupu pro každou alokaci/uvolnění objektu je toto prováděno dávkově – je alokováváno větší množství[15] objektů v rámci jedné CAS operace a tyto jsou udržovány ve struktuře vyhrazené pro jediné jádro (tj. bez nutnosti zajistit výlučný přístup). Přístup do hlavní fronty je vyžadován pouze tehdy, pokud je cache plná (není možné uvolňovat další objekty) nebo prázdná (není možné alokovat další objekty). Mimo automatických cache pro každé jádro je možné vytvářet uživatelské cache, které fungují stejným způsobem.

[12] standardní knihovna C
[13] z angl. Memory Pool
[14] Non-Uniform Memory Access - přístup do různých částí paměti z různých jader trvá různě dlouho
[15] určeno nastavením konstanty CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE při překladu

### 2.3.4 Network Packet Buffer Management

Knihovna `librte_mbuf` implementuje strukturu `mbuf`[16], která je používána DPDK aplikacemi pro uchovávání paketů (ale je možné ji využít pro uložení libovolných dat). Počet mbufů, jejich formát i velikost jsou pevně dané. Všechny jsou vytvořeny při startu aplikace a uloženy v `mempool`. Hlavička mbufu je vývojáři dlouhodobě držena na minimální možné velikosti – v současnosti pouze dva řádky cache[17], tedy 128 bajtů. Nejčastěji používané položky se nacházejí v první polovině hlavičky, takže mnohdy není třeba přesouvat do cache druhý řádek[3].




Obrázek 2.1: Struktura mbuf

**Význačné prvky hlavičky[4]:**

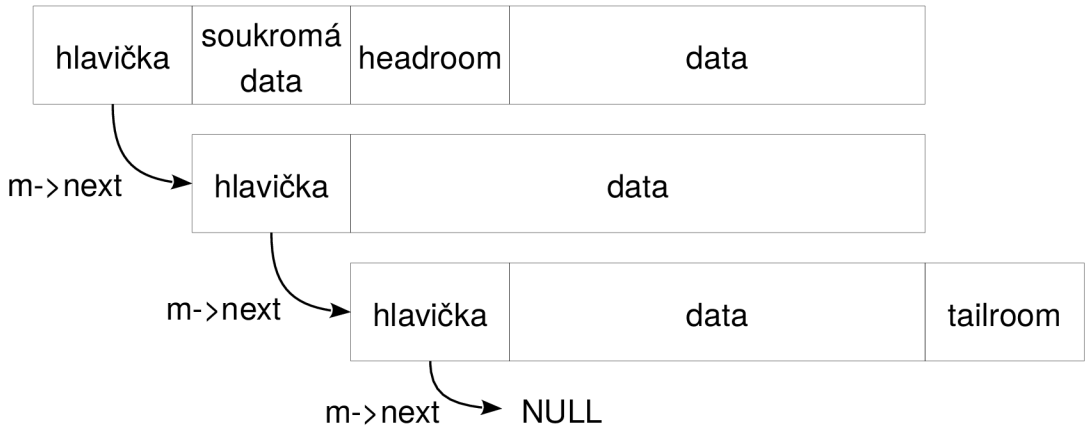
- next – ukazatel na další segment
- nb_segs – počet segmentů
- buf_addr – ukazatel na začátek datové oblasti
- priv_size – velikost privátních dat aplikace
- data_len – délka dat v tomto mbufu
- pkt_len – délka paketu (součet délek všech segmentů)
- port – číslo portu
- pool – ukazatel na mempool, ze kterého mbuf pochází

Začátek oblasti pro data (nazýváno headroom[18]) zůstává volný pro využití aplikací. Ta tak může vložit například protokolové hlavičky před začátek paketu, aniž by bylo nutné provádět posun dat paketu. Samotný začátek dat je pro optimální výkon vždy zarovnán na řádky cache. Druhou zvažovanou reprezentací bylo oddělení kontrolních dat (hlavičky) a dat paketu do dvou oddělených membufů. Nakonec byl z výkonnostních důvodů (pro alokaci stačí jediná operace) zvolen tento formát.

[16] Message Buffer
[17] míněno moderních CPU Intel
[18] velikost je možné určit při překladu nastavením konstanty RTE_PKTMBUF_HEADROOM

### Řetězení mbufů

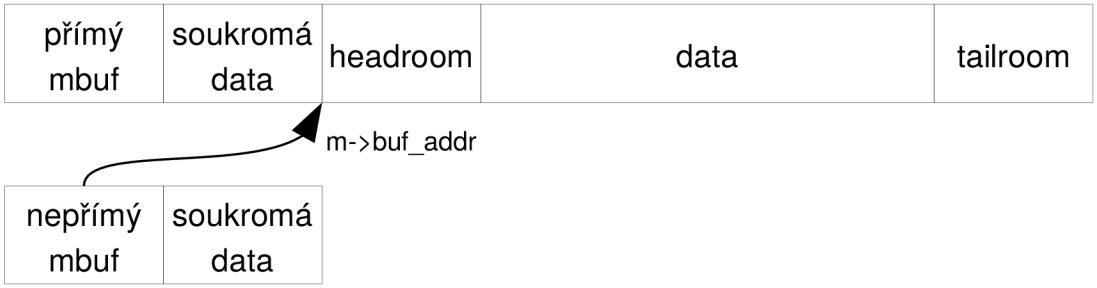
Pokud velikost paketu přesáhne kapacitu mbufu (například jumbo[19] rámce), může jich být několik zřetězeno (ty se pak nazývají segmenty). Hlavička mbufu v takovém případě obsahuje ukazatel na další mbuf a následující mbufy již neobsahují meta informace. Při uvolňování mbufu jsou pak uvolněny všechny. Uvolnění mbufu znamená, že je vrácen do mempoolu, ze kterého byl alokován[3].




Obrázek 2.2: Řetězení mbufů

**Přímé a nepřímé mbufy** Pro účely duplikace a fragmentace paketů existují tzv. nepřímé mbufy. Oproti přímým (běžným) neobsahují data a pouze odkazují na přímý mbuf – tímto způsobem je možné použít vícekrát stejná paketová data bez nutnosti je kopírovat. Každý mbuf má čítač referencí, který je s každým navázáním nepřímého mbufu inkrementován a později dekrementován při jeho odvázání. Přímý mbuf je uvolněn, pokud je výsledná hodnota čítače 0.

Mbuf se může stát nepřímým, pouze pokud se na něj neodkazuje jiný nepřímý mbuf. Stejně tak není možné, aby nepřímý mbuf odkazoval na jiný nepřímý mbuf (namísto toho se automaticky vytvoří reference na příslušný přímý mbuf). Pro používání nepřímých mbufů je vhodné vytvořit samostatný mempool s výrazně sníženou velikostí mbufů (neobsahují data) a dosáhnout tak nižšího využití paměti[3].




Obrázek 2.3: Přímý a nepřímý mbuf

[19]Ethernet rámce s velikostí přes 1500 B

### 2.3.5 Ovladače síťových zařízení

DPDK již obsahuje ovladače pro mnoho běžných síťových karet (zejména PCI-E karty). Tyto ovladače využívají tzv. Ethernet Device API[20], které musí ovladače implementovat. Na rozdíl od běžných ovladačů, které jsou součástí jádra OS, jsou ovladače DPDK implementovány výhradně v uživatelském prostoru a pro jejich vývoj je možné používat celý jazyk C včetně standardní knihovny (jaderné ovladače mají jistá omezení). Kromě povinných funkcí pro příjem, odesílání, inicializaci a konfiguraci zařízení obsahuje API také funkce pro zobrazování statistik, informací a pro filtrování.

Síťové zařízení je reprezentováno strukturou `rte_eth_dev`, jejíž součástí jsou ukazatele na data zařízení (`rte_eth_dev_data`), funkce pro příjem a odesílání. Dále struktura `rte_dev_ops`, která obsahuje ukazatele na funkce implementující další operace. DPDK ovladače typicky nevyužívají systém přerušení a přítomnost paketů je kontrolována v aktivní smyčce (tzv. polling) – z tohoto důvodu se jim říká PMD (Poll Mode Driver). Výhodou je zejména odstranění režie přerušení (vyjma upozornění na změnu stavu linky), která výrazně omezuje výkon při vysokém množství paketů. Nevýhodou je vyšší spotřeba při nízké či žádné zátěži[3].

**Modely zpracování paketů** DPDK umožňuje použití dvou modelů – synchronního run-to-completion a asynchronního pipe-line:

- run-to-completion – jedno jádro zabezpečuje příjem, zpracování i odesílání paketů jedné fronty
- pipe-line (zřetězené zpracování) – jedno jádro přijímá pakety z jedné nebo více RX front. Tyto pakety jsou pak skrze Ring předány ostatním jádrům ke zpracování a případnému odeslání

V závislosti na požadavcích aplikace na latenci paketů je možné nastavit velikost shluků paketů, které jsou přijímány/odesílány v rámci jednoho volání. Větší shluky přinášejí vyšší efektivitu snížením počtu volání funkcí (každé volání je spojeno s určitou režií), ale způsobují vyšší latenci (aplikace dostává paket později, než by bylo možné).

Při přijímání paketu v PMD jsou vyplněny určité položky mbufu a při odesílání jsou na základě informací v mbufu nastaveny odesílací deskriptory. Některá síťová zařízení podporují HW akceleraci časově náročných operací – například počítání kontrolních součtů IPv4 nebo L4 hlaviček. Toto PMD indikuje aplikaci a zároveň se stará o korektní vyplnění struktury mbuf.

Pro testovací a ladící účely je možné používat virtuální síťová zařízení (bez reálného HW), které mohou k přijímání/odesílání paketů využívat například soubory typu PCAP.

[20] Ethernet Device API definuje sadu funkcí pro práci se zařízením a pakety

# Kapitola 3

# HW platforma

Pro implementaci a testování 10 Gb Ethernet MAC jsem měl k dispozici vývojovou platformu Reflex CES Achilles Arria 10 SoC SoM s PCIe kartou osazenou SFP+[1] sloty pro 10G optické transceivery. Jako operační systém slouží modifikovaná verze Linuxu.

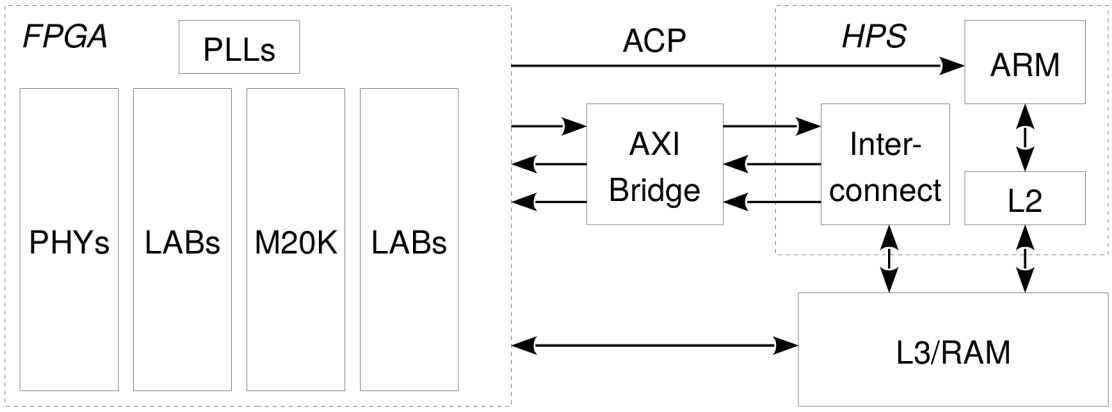
**FPGA** (Field Programmable Gate Array) je čip, který umožňuje implementaci široké škály uživatelských obvodů zapsaných v HDL[2] a přeložených vhodným syntézním nástrojem (např. Vivado nebo Quartus). Čip je tvořen polem programovatelných logických bloků a sítí konfigurovatelných propojů, které jsou použity pro implementaci požadovaného obvodu. Kromě obecných logických bloků (např. LUT) obsahuje typicky také specializované obvody (DSP, PHY či DDR RAM řadič) a bloky dedikované paměti[17]. Obrovskou výhodou je možnost dynamické rekonfigurace čipu, nevýhodou je nižší dosažitelný výkon (pouze stovky MHz) oproti výkonům, které je možné dosáhnout s ASIC.

**SoC** (System on Chip) je integrovaný obvod, který integruje mnoho různých komponent (obvykle vše, co je třeba pro funkční počítač). Typické součásti jsou CPU, řadiče, paměť a V/V porty, ale může obsahovat i další obvody rozšiřující funkcionalitu celého systému (tímto obvodem může být i FPGA). Jednotlivé prvky jsou propojeny širokou škálou sběrnic (dle jejich potřeb) a vzájemně mezi sebou komunikují. SoC kombinující běžné CPU a FPGA může být využito pro širokou škálu úkolů – například pro pokročilou analýzu síťového provozu, kdy masivně paralelní obvod v FPGA poskytuje potřebnou hrubou sílu a CPU se stará o ukládání a zobrazování výstupů FPGA či o specifickou analýzu malého množství paketů.

## 3.1 Arria 10 SoC

Arria 10 SoC je platforma kombinující CPU se dvěma jádry ARM Cortex-A9 a FPGA Intel Arria 10. Zaměřuje se hlavně na energetickou efektivitu a poměr cena/výkon. Produktová řada se nazývá Arria 10 SX a kromě běžných vlastností FPGA nabízí také rychlé (až 17,4 Gb/s) sériové transceivery, které je možné použít pro implementaci většiny sériových protokolů – například PCIe Gen3, SATA nebo 10 Gb Ethernet[8]. Tato schopnost je ve spojení s integrovaným PHY (nazýváno Enhanced PCS) klíčová pro efektivní implementaci Ethernet rozhraní[10], které je vytvářeno v rámci této práce.

[1] Small Form-factor Pluggable – standard pro připojování transceiverů různých výrobců
[2] Hardware Description Language




Obrázek 3.1: Platforma Arria 10 SoC

Základním stavebním blokem FPGA Arria 10 jsou LAB (logic array block). Ty jsou poskládané z ALM (adaptive logic modules) a v obvodu implementují logické nebo aritmetické funkce a registry. Čtvrtina z nich může být také využita jako MLAB (memory LAB) pro implementaci malých pamětí (kapacita jedné MLAB je 640 bitů). Pro velké paměti jsou k dispozici dedikované M20K bloky (velikost 20 Kb). Dále jsou k dispozici různé hard IP core – například sčítačky, násobičky a zejména HSSI[3], PHY a DDR4 řadič. Hodinové signály jsou generovány interními PLL tří různých typů (k dispozici jsou celočíselné i frakční)[8].

Hlavní součástí HPS (Hard Processor System) je MPU (Microprocessor Unit) subsystémem, který obsahuje ARM jádra, L2 cache a moduly pro ladění. Dále jsou přítomny řadiče USB, tři 1 GbE MAC pro přímou komunikaci s CPU a mnoho dalších komponent. Vzájemnou komunikaci zajišťuje propojovací logika nazvaná Main L3 Interconnect. Komunikace mezi HPS a FPGA probíhá skrze dvě AXI a jedno AXI-Lite rozhraní ovládané komponentou AXI Bridge, která umožňuje propojování různých šířek AXI. Samostatné je AXI rozhraní s podporou ACP a také připojení k L3 a RAM řadiči, které jsou k dispozici pro HPS i FPGA[9].

## 3.2 DMA

DMA (Direct Memory Access) je způsob přenosu dat mezi pamětí a I/O zařízením (Ethernet rozhraní v našem případě), kdy data nepřenáší přímo CPU, ale jsou kopírována samostatným kanálem (AXI v případě Arria 10). Toto kopírování je řízeno DMA řadičem, který dostává od aplikace adresy, na které mají být příchozí pakety zapsány a potvrzuje ukončení přenosu (v tu chvíli může klientská aplikace začít pracovat s daným paketem).

Tento přístup výrazně zvyšuje výkon, ale přináší problémy se zajištěním koherence vyrovnávacích pamětí CPU. Při DMA zápisu do hlavní paměti je třeba informovat vyrovnávací paměti, jinak hrozí, že CPU přečte neplatnou hodnotu. Při DMA čtení vzniká nebezpečí, že data zapsána CPU nebudou včas propagována z vyrovnávací do hlavní paměti a tedy budou neaktuální. Naše platforma nabízí (ACP) rozhraní pro zajištění jednostranné koherence, konkrétně pro zápisy z FPGA do hlavní paměti. Toto rozhraní bude využíváno pro synchronizaci DMA řadiče v FPGA a klientské DPDK aplikace běžící na CPU.

[3]High-speed serial interface

# Kapitola 4

# Návrh 10 Gb Ethernet rozhraní

Návrh byl vytvářen na základě standardu IEEE 802.3, požadavků a prostředků platformy a s cílem co nejnižšího využití zdrojů FPGA. V zájmu omezení potřebných zdrojů byly do návrhu zahrnuty jen povinné vlastnosti a dále vlastnosti požadované v rámci projektu SProbe. MAC nepodporuje systém přerušení a funguje výhradně v dotazovacím režimu (polling). Technickou konzultaci a pomoc poskytoval Ing. Vlastimil Košař. Jako DMA řadič byla použita již existující komponenta vytvořená Bc. Michalem Orsákem v rámci práce na VUT FIT.

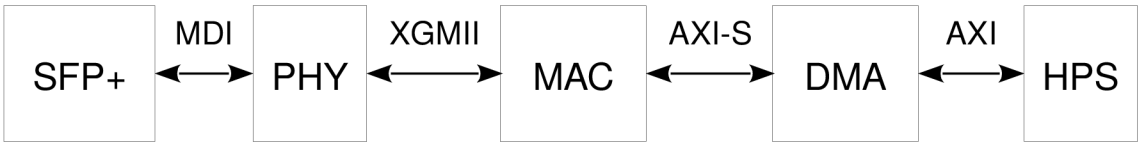
Část komponent koncepčně vychází z předchozí práce na 1 GbE MAC v rámci předmětu Projektová praxe a část zdrojových kódů je převzata (zejména pomocné komponenty). Na začátku vývoje 10 GbE MAC pokračovala spolupráce s bývalým studentem Filipem Brázdou a některé jeho komponenty v práci zůstaly (se značnými úpravami). Jako AXI-Lite Endpoint byla se svolením použita již existující komponenta vytvořená Ing. Janem Viktorinem.

**Poskytované vlastnosti 10GbE MAC:**

- full duplex
- volitelná simplex konfigurace (přítomna jen TX či jen RX část) pro úsporu zdrojů
- AXI Lite rozhraní pro změnu konfigurace za běhu
- promiskuitní režim (příjem rámců s libovolnou MAC adresou)
- volitelná filtrace příchozích paketů na základě MAC adresy (až 7 adres)
- nastavitelné odstranění FCS příchozích rámců
- statistické čítače (total, bytes, error, dropped)
- FIFO rozhraní s délkami příchozích rámců
- volitelný MDIO řadič
- kernel ovladač pro konfiguraci
- DPDK ovladač pro konfiguraci a příjem/odesílání paketů

## 4.1 Popis 10 Gb Ethernet rozhraní

Ethernet rozhraní je kombinace HW a SW komponent splňujících požadavky protokolu Ethernet, jak jsou definovány v standardu IEEE 802.3[7]. Programová část (ovladač) řídí konfiguraci HW komponent a dále komunikaci mezi HW a operačním systémem. Hardware se typicky skládá ze tří komponent – PHY, MAC a DMA. PHY (Physical Layer Device) zajišťuje dodržení specifikací přenosového média a formátu přenášených dat (zejména pravidel kódování dat). MAC slouží jako rozhraní mezi fyzickou a linkovou vrstvou a je obvykle připojen k DMA řadiči, který ukládá příchozí rámce do hlavní paměti a načítá z ní rámce určené k odeslání.




Obrázek 4.1: Schéma 10GbE rozhraní na platformě Arria 10

Prvním článkem rozhraní je SFP+ slot, do kterého je možné dle potřeby vložit konkrétní (obvykle optický) transceiver. Ten pak zajišťuje převod optického signálu na elektrický, který je přenášen MDI[1] a zpracováván ve PHY. Implementace PHY může být realizována jako samostatný čip, obvod v FPGA (tzv. soft IP core) nebo nejčastěji jako součást SoC (tzv. hard IP core). Poslední možnost je nejvýhodnější z hlediska výkonu (výkon FPGA obvodů je limitován) a ceny (PHY v FPGA zabere část zdrojů – může být nutné pořídit větší FPGA).

Propojení MAC a PHY vrstev zajišťuje XGMII[2] rozhraní. Mapování XGMII na interní signály MAC poskytuje RS (Reconciliation Sublayer), která je povinnou součástí MAC[7]. RS také zajišťuje nepřetržité odesílání idle znaků (či chybových znaků v případě chyby) a kontroluje výskyt chybových znaků v příchozích datech.

Komunikace mezi MAC a DMA probíhá sériově přes AXI Stream (AXI-S) rozhraní, přičemž jeden DMA řadič může typicky obsluhovat více MAC kanálů. Komponenta HPS (Hard Processor System) reprezentuje hard IP core, který je součástí SoC a obsahuje CPU jádra, paměťový řadič a propojovací logiku pro komunikaci mezi FPGA a CPU/pamětí.

### MAC

Hlavní HW částí 10Gb Ethernet (dále 10GbE) rozhraní je Ethernet MAC[3], který zajišťuje dodržování specifikace protokolu (například zajištění 12B mezery mezi rámci), vkládání preambule a výpočet/kontrolu kontrolního součtu rámce (CRC). Každý MAC je identifikován MAC adresou (také nazývána "fyzická adresa"), která je jedinečná a obvykle přidělována při výrobě. Některé MAC ale umožňují adresu nastavovat anebo se identifikovat více adresami. Ve výchozím režimu přijímá MAC pouze rámce s odpovídající cílovou MAC adresou. Při zapnutí tzv. promiskuitního režimu jsou pak přijímány všechny rámce.

[1] Medium Dependent Interface
[2] 10 Gigabit Media Independent Interface
[3] Medium Access Controller

### Protokol Ethernet

Ethernet definuje tvar datového paketu na linkové vrstvě. Ten vzniká zabalením paketu protokolu vyšší vrstvy (typicky IP) a je nazýván Ethernet frame (rámec). Přidáním preambule a SFD[4] pak vzniká Ethernet paket. V tomto tvaru je pak paket odesílán ke zpracování v PHY a odeslání.

Preambule historicky slouží k synchronizaci hodinových signálů PHY odesílatele a příjemce a skládá se z 56 alternujících bitů (hodnoty 1 a 0). SFD následuje ihned po preambuli a řadu alternujících bitů zakončuje dvojicí bitů hodnoty 1. Tímto je označen začátek Ethernet rámce – dvojice MAC adres identifikující příjemce a odesílatele. Před samotnými daty paketu vyšší vrstvy je uvedena jeho délka. Maximální délka pro běžný Ethernet rámec je 1500 bajtů, ale síťová zařízení často podporují i větší délky – tzv. jumbo rámce. Využití těchto nestandardních rámců může být problematické, ale vede ke zvýšení efektivní propustnosti (menší část pásma je využita hlavičkami) a potencionálnímu snížení režie ze strany CPU (je přijímáno méně paketů).

FCS[5], neboli kontrolní součet rámce, slouží k zachycení chyb během přenosu a je počítán nad chráněnými poli Ethernet rámce – všemi kromě FCS, preambule a SFD. Pokud přijatá data neodpovídají přijatému FCS, celý rámec je zahozen. Pro výpočet FCS je použita varianta kódu CRC[6] nazývaná CRC-32. Poslední částí Ethernet paketu je mezipaketová mezera, která má poskytovat příjemci čas pro přípravu na přijetí následujícího paketu. Minimální mezipaketová mezera je dlouhá 12 bajtů, maximální délka není omezena.

Tabulka 4.1: Ethernet rámec

| **Pole** | **Délka** |
|---|---|
| Preambule | 7 oktetů |
| SFD (Start Frame Delimiter) | 1 oktet |
| Cílová MAC adresa | 6 oktetů |
| Zdrojová MAC adresa | 6 oktetů |
| Délka dat | 2 oktety |
| Data (payload) | 46-1500 oktetů |
| FCS (Frame Check Sequence) | 4 oktety |
| Mezipaketová mezera | 12 oktetů |

### DIC (Deficit Idle Count)

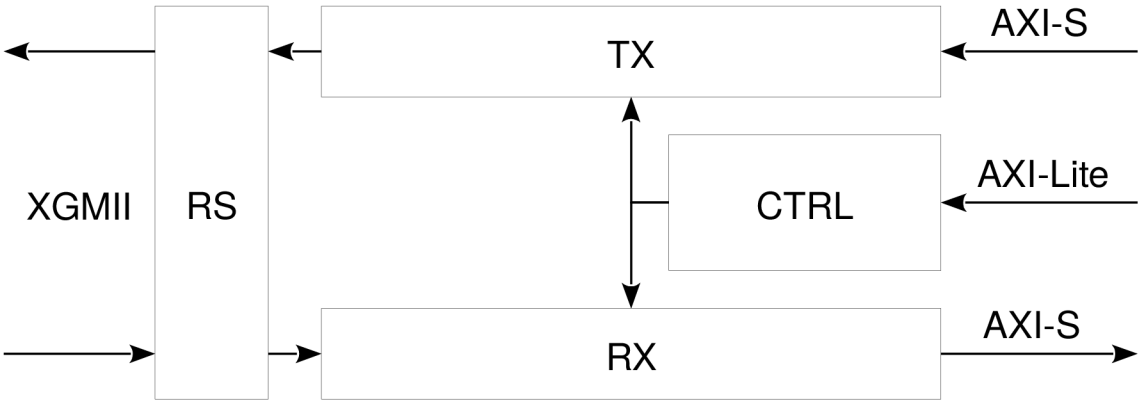
10Gb Ethernet vyžaduje, aby byl počáteční znak vždy zarovnán na linku 0 – tedy první bajt slova přenášeného na XGMII rozhraní (viz kapitola 5.1). Toho je možné dosáhnout vkládáním idle bajtů navíc, což ovšem znamená, že mezera mezi jednotlivými rámci bude delší než 12 bajtů. Nestandardně dlouhá mezera ovšem oproti specifikaci sníží maximální propustnost 10GbE rozhraní.

Řešením tohoto problému je implementace DIC[14], kdy je sledována délka mezer mezi předchozími pakety a idle bajty jsou přidávány a odebírány dle potřeby (reálná délka mezery se pohybuje mezi 9 a 15 bajty). Průměrná délka mezery tak zůstává standardních 12 bajtů a propustnost odpovídá specifikaci.

[4] Start Frame Delimiter
[5] Frame Check Sequence
[6] Cyclic Redundancy Check

## 4.2 Návrh MAC

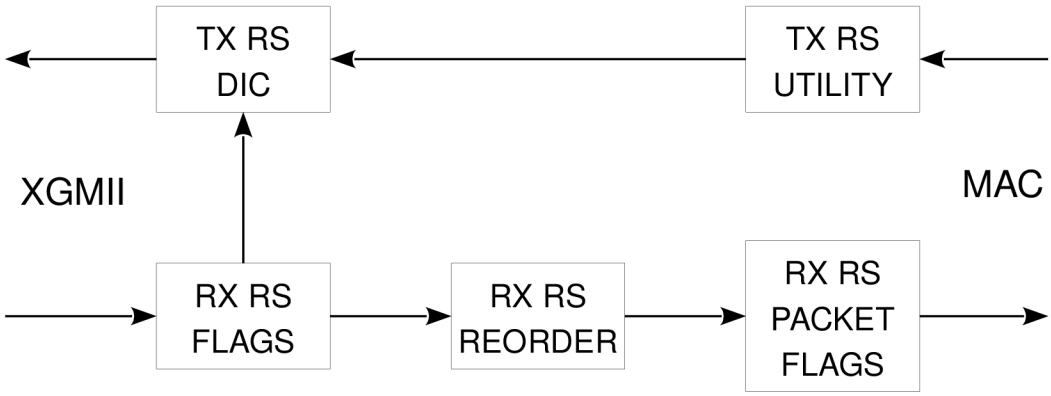
MAC byl navržen s ohledem na modularitu a maximální jednoduchost. MAC je možné používat v konfiguraci pouze TX nebo pouze RX, zatímco RS a CTRL[7] jsou vždy přítomné a aktivní. CTRL řídí aktivitu RX a TX a obsahuje statistické čítače, stavový registr a registry pro konfiguraci. K těmto přistupuje ovladač (DPDK nebo jaderný) skrze AXI-Lite sběrnici.




Obrázek 4.2: Schéma 10GbE MAC

### Subsystém RS

Reconciliation Sublayer pro XGMII je implementována dvěma komponentami na vysílací straně a třemi na přijímací. RX RS FLAGS detekuje chybové stavy na lince a informuje o nich TX RS DIC, která adekvátním způsobem mění formát odchozích dat.




Obrázek 4.3: Schéma RS

**RX RS FLAGS** Detekuje v příchozích datech kontrolní znaky – START, TERMINATE (konec paketu), LOCAL FAULT (tato chyba přichází, pokud byl problém detekován v lokálním PHY) a REMOTE FAULT (tato chyba je generována RS na druhém konci spojení při detekci chyby v PHY).

[7] Control

**RX RS REORDER** Zajišťuje zarovnání kontrolního znaku START na první bajt (viz kapitola 5.1).

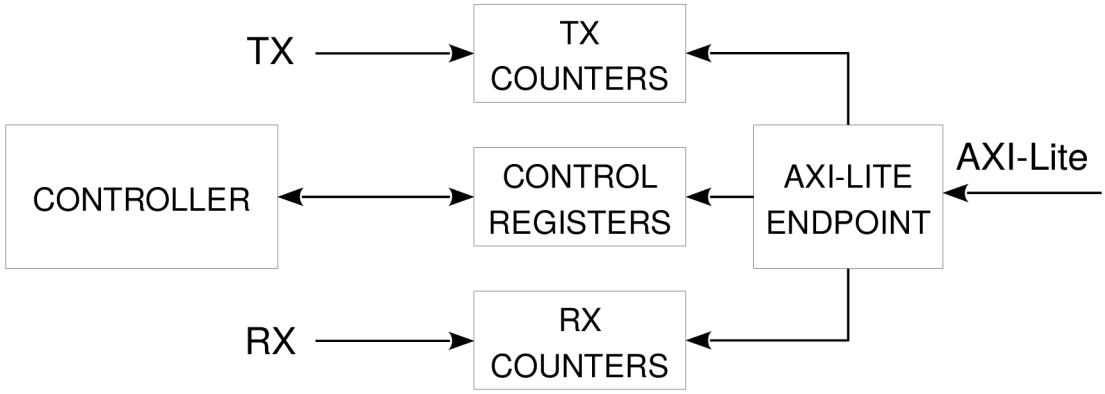
**RX RS PACKET FLAGS** Převádí XGMII na interní rozhraní MAC – generuje signály TKEEP (popisuje, které bajty obsahují platná data), SOF (Start of Frame), EOF (End of Frame), ERROR (indikující chybu přenosu) a VALID.

**TX RS UTILITY** Vkládá za poslední bajt paketu kontrolní znak TERMINATE dle potřeby následovaný IDLE znaky – vzdálené RS vyžaduje ukončovací znak pro identifikaci konce paketu.

**TX RS DIC** Konvertuje interní rozhraní MAC na XGMII a zajišťuje vložení správně dlouhé mezipaketové mezery (dle pravidel DIC). Dále reaguje na stav LOCAL FAULT odesíláním kontrolních znaků REMOTE FAULT (tím je vzdálená RS informována o problému se spojením) a na stav REMOTE FAULT přerušením odesílání paketů (tj. čeká se na vyřešení problému ve vzdálené PHY).

### Subsystém CTRL

Řízení MAC zajišťuje CONTROLLER, který také zapisuje informace o stavu do stavového registru a reaguje na změny konfiguračních registrů. TX a RX inkrementují příslušné statistické čítače a všechny registry jsou zpřístupněny ovladači na AXI-Lite rozhraní jednotkou AXI-LITE ENDPOINT.




Obrázek 4.4: Schéma CTRL

**CONTROLLER** Řadič se stará o korektní průběh restartu komponenty (při spuštění a na základě požadavku z ovladače), vypínání/zapínání komponent (na základě konfigurace) a ovládání filtru příchozích paketů.

**AXI-LITE ENDPOINT** Implementuje tzv. slave[8] rozhraní AXI-Lite – vytváří adresový prostor obsahující jednotlivé registry. SW může posílat požadavky na čtení nebo zápis a zjišťovat tak stav MAC či měnit jeho konfiguraci.

[8] podřízená strana komunikace – odpovídá na příkazy

**CONTROL REGISTERS**

- **Řízení a stav** – obsahuje informace o stavu, schopnostech (např. přítomnost TX a RX nebo maximální počet MAC adres) a řídící bity pro restart jednotky a zapínání/vypínání funkcí (např. promiskuitní režim)
- **MAC adresa** – umožňuje vkládání MAC adres pro filtrování příchozích paketů
- **Řízení MDIO** (volitelně) – pro komunikaci po sběrnici MDIO[9] (typicky pro řízení externího PHY čipu)

**TX COUNTERS**

- Celkový počet paketů
- Počet zahozených paketů
- Počet bajtů

**RX COUNTERS**

- Stejné čítače jako TX
- Počet chybných paketů

**Subsystém TX**


Obrázek 4.5: Schéma TX

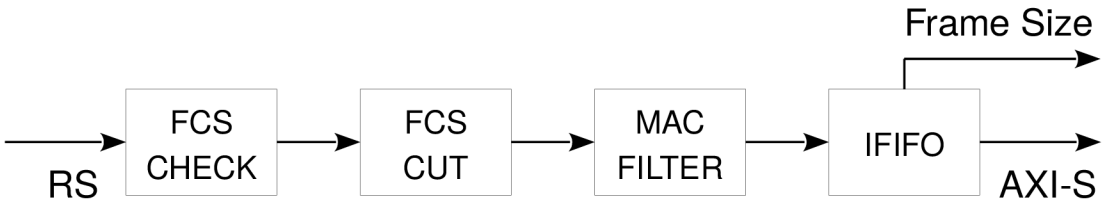
**FCS GEN** Převádí vstupní AXI Stream na interní rozhraní MAC (generuje SOF signál). Dále provádí kontrolní součet na každém slovu odesílaných dat a vkládá výsledek (FCS) na konec paketu.

**HEADER GEN** Přidává před paket hlavičku Ethernet paketu (preambule a SFD).

**OFIFO** Output FIFO implementuje frontu pro odesílané pakety s architekturou store-and-forward[10]. Oproti alternativní architektuře cut-through[11] zvyšuje latenci, ale je obecně spolehlivější (XGMII vyžaduje, aby byl přenos uskutečněn bez přerušení). Pokud by tedy z nějakého důvodu došlo ke zpoždění části odesílaného paketu (například kvůli vysokému zatížení rozhraní DMA-RAM), celý odesílaný paket je při využití cut-through nutné zahodit. Kvůli store-and-forward je omezena maximální velikost odesílaného paketu – pokud přesáhne celkovou velikost OFIFO, je paket zahozen. Při odeslání/zahození paketu jsou patřičně inkrementovány TX čítače.

[9] Management Data Input/Output
[10] ulož a pošli dál – před odesláním paketu na XGMII je nejprve celý uložen do fronty
[11] paket je odesílán okamžitě po přijetí jeho prvních částí

### Subsystém RX




Obrázek 4.6: Schéma RX

**FCS CHECK** Provádí výpočet CRC-32 na příchozích datech a ověřuje tak bezchybnost přijímaného paketu. Pokud FCS nesouhlasí, je signalizována chyba a celý paket je zahozen. Kromě kontroly správnosti paketu také počítá jeho délku.

**FCS CUT** Odřezává FCS (typicky je nežádoucí, aby zůstalo) z konce Ethernet rámce. Tuto jednotku je ale možné vypnout (např. pro potřeby analýzy síťového provozu) a FCS ponechat.

**MAC FILTER** Kontroluje shodu cílové MAC adresy příchozího paketu s některou z uživatelem vložených MAC adres. Vypnutím této komponenty se MAC přepíná do promiskuitního stavu – jsou přijaty všechny pakety bez ohledu na jejich cílovou MAC adresu.

**IFIFO** Input FIFO implementuje frontu pro příchozí pakety. Před odesláním paketu na AXI Stream rozhraní se čeká na jeho kompletní uložení a také na výsledek kontroly FCS a cílové MAC adresy. Pro potřeby některých projektů je přidáno FIFO rozhraní obsahující délky přijatých rámců (další komponenty tedy nemusí znovu počítat délku rámce). S koncem každého paketu jsou adekvátně inkrementovány čítače.

### Realizace některých vlastností MAC

**Simplex** 10 Gigabit Ethernet je definován pouze pro plně duplexní režim. Z toho důvodu zůstává při použití nestandardního simplex režimu aktivní celá RS a vypnutý kanál se chová stejně jako nevyužitý kanál. Toto je důležité zejména z hlediska zpracování chybových stavů, kdy protistrana očekává jisté odpovědi. Jakékoliv příchozí rámce jsou v režimu TX Simplex vždy zahazovány.

**MDIO** Samostatným subsystémem je MDIO řadič, který je možné v případě potřeby aktivovat a využívat pro řízení některých externích PHY čipů, ale také libovolných dalších komponent s MDIO rozhraním. Komponenta čte registr připojený k AXI-Lite a vykonává čtecí nebo zápisové operace na základě hodnot v něm. Výsledek operací je pak uložen do stejného registru. MDIO využívá datový signál o šířce jeden bit, takže je třeba jakékoliv přenosy serializovat a deserializovat. K tomu jsou využity posuvné registry řízené FSM.

# Kapitola 5

# Implementace

Implementace HW komponent byla provedena v jazyce VHDL[1] a jako nástroj pro syntézu byl použit Intel Quartus Prime. Oba ovladače (DPDK a jaderný) pro OS Linux byly implementovány v jazyce C. Kompilaci zajistil open-source balík překladačů GCC.

## 5.1 Popis použitých rozhraní

Některá implementační rozhodnutí přímo souvisí s pravidly použitých rozhraní. V této části je proto uveden jejich stručný popis.

### MDI

Medium Dependent Interface (rozhraní závislé na přenosovém médiu) je rozhraní mezi PHY a přenosovým médiem. Existuje poměrně velké množství různých MDI a každé je definováno příslušným standardem (např. 10GBASE-R). O propojení MDI a MII[2] se stará PHY a při implementaci MAC tedy není třeba jeho existenci řešit[7].

### XGMII

10 Gigabit Media Independent Interface (XGMII) je typ MII používaný pro připojení 10 GbE MAC k různým typům PHY. Jedná se o synchronní DDR[3] rozhraní s nezávislými 32 bitovými datovými a kontrolními cestami pro RX a TX. Každý směr sestává ze tří signálů – hodinového, kontrolního a datového. XGMII podporuje pouze plně duplexní režim a obě strany komunikace musí generovat nepřetržitý proud dat nebo kontrolních znaků. Datový proud je tvořen mezipaketovou mezerou (v jejímž rámci probíhá signalizace chyb), Ethernet preambulí, SFD[4], samotnými daty a EFD[5].

Datové signály jsou organizovány do 4 linek:

| **Bity:** | 0 až 7 | 8 až 15 | 16 až 23 | 24 až 31 |
|---|---|---|---|---|
| **Linka:** | 0 | 1 | 2 | 3 |

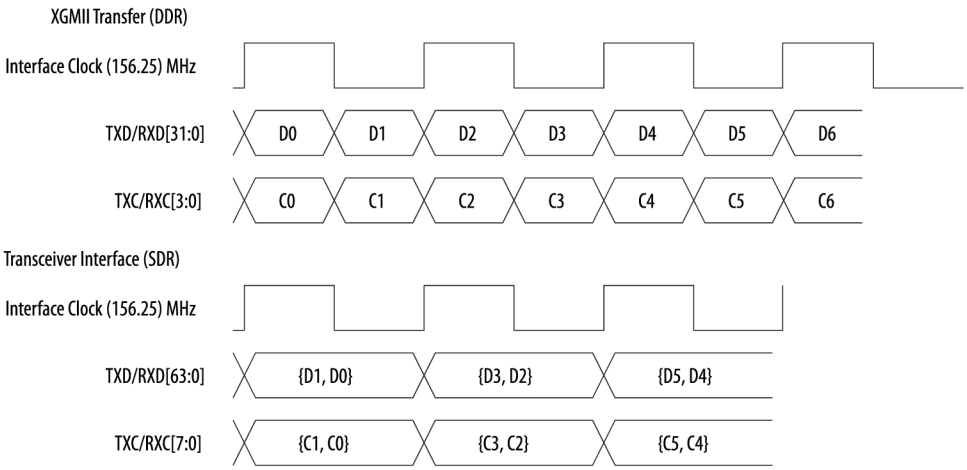
Tabulka 5.1: XGMII datový kanál

[1] VHSIC Hardware Description Language
[2] Medium Independent Interface
[3] Double Data Rate – přenos dat se odehrává na nástupné i sestupné hraně hodinového signálu
[4] Start Frame Delimiter
[5] End Frame Delimiter

Specifikace XGMII stanovuje, že frekvence hodinového signálu je rovna podílu přenosové rychlosti 10 GbE a počtu bitů přenesených za jednu periodu (64[6]) – tedy 156,25 MHz[7]. Dále je definováno, že SFD musí být přenášen výhradně na XGMII lince 0.

Kontrolní signál je široký 4 bity – jeden bit pro každou XGMII linku – a značí přítomnost dat (kontrolní bit má hodnotu 0) nebo kontrolního znaku (kontrolní bit má hodnotu 1). Významné kontrolní znaky jsou SFD, EFD, IDLE a signalizace chyb LOCAL FAULT a REMOTE FAULT. Během přenosu dat je výskyt hodnoty 1 v kterémkoliv z kontrolních bitů vyhodnocen jako chyba a přenášený paket je zahozen.

PHY dostupné v čipu Arria 10 nepodporuje standardní DDR XGMII rozhraní[10]. Namísto něj je podporována SDR[7] verze XGMII a stejné propustnosti je dosaženo zdvojnásobením šířky datového signálu na 64 bitů. Toto je běžná praxe i na srovnatelných platformách (Xilinx) a přináší jisté výhody – například není nutné převádět DDR XGMII rozhraní na interní SDR sběrnici MAC. Na následujícím obrázku je znázorněna rozdílná podoba stejné transakce.




Obrázek 5.1: Rozdíl mezi DDR a SDR XGMII (převzato z [10])

### AXI

AXI (Advanced Extensible Interface) je součást otevřeného standardu ARM AMBA[8], který definuje[13] rodinu sběrnic pro komunikaci v rámci čipu (mezi jednotlivými částmi SoC). V rámci práce jsou využívány následující verze:

- **AXI3** – pro memory-mapped[9] transakce s požadavkem na vysoký výkon
- **AXI4-Lite** – pro jednoduchou memory-mapped komunikaci s nízkým výkonem (například konfigurace MAC nebo čtení statistických čítačů)
- **AXI4-Stream** – pro přenos proudů dat mezi komponentami v rámci FPGA

[6] přenos probíhá na obou hranách hodinového signálu – během periody tedy dojde ke dvěma 32b přenosům
[7] Single Data Rate
[8] Advanced Microcontroller Bus Architecture
[9] registry FPGA komponent mají přiřazenu adresu a přístupy do hlavní paměti jsou také prováděny na základě adresy

### AXI3

AXI3 je využíváno pro přenos paketových dat mezi DMA řadičem a hlavní pamětí. K dispozici jsou až tři samostatné AXI kanály o šířce 64 bitů a jeden AXI kanál s podporou ACP, rovněž o šírce 64 bitů. Mezi hlavní vlastnosti protokolu patří oddělené fáze přenosu adresy/příkazu a samotných dat, podpora dávkových operací (jeden přenos adresy a mnoho datových přenosů) a oddělené kanály pro zápisové a čtecí operace, což umožňuje dosáhnout nízké režie DMA přenosů[13].

**ACP** 64b AXI rozhraní s podporou ACP je možné použít pro přímý, koherentní přístup (čtení i zápis) do L2/L3 cache[13]. Toto rozhraní je využíváno DMA řadičem pro přenos deskriptorů a délek paketů. DMA může takto přistupovat ke cache systému stejně jako ARM Cortex-A9 jádra a tak zvýšit výkon celého řešení – přenos metadat mezi CPU (resp. ovladači na něm běžícím) a DMA většinou neprochází přes výrazně pomalejší RAM. Pokud cache požadovaná data neobsahuje, jsou čtena z hlavní paměti.

### AXI-Lite

Pro přístup do konfiguračních a statistických registrů MAC i DMA řadiče je používána tato odlehčená varianta AXI rozhraní. Šířka datového signálu byla omezena na 32 nebo 64 bitů, všechny transakce využívají plnou šířku rozhraní a došlo k odstranění podpory dávkového přenosu[13]. Způsob použití je ale podobný běžnému AXI a je dokonce možné propojit plnohodnotné AXI a AXI-Lite (propojovací logika v případě potřeby převodu je obvykle vkládána automaticky použitým nástrojem pro syntézu).

### AXI-Stream

Nejčastější způsob připojení je master-slave, kdy master generuje data a slave je přijímá a jejich přijetí potvrzuje. Proces potvrzení (handshake) je jednoduchý - nadřízená komponenta nastaví signál TVALID do hodnoty 1 a udržuje na sběrnici platná data až do nastavení signálu TREADY podřízenou komponentou[12]. V rámci proudu dat se mohou vyskytovat datové nebo nulové bajty (datový bajt je značen nastavením příslušného bitu signálu TKEEP do hodnoty 1), ale pro zjednodušení obvodu povoluje MAC nulové bajty pouze v posledním slově Ethernet rámce a datové bajty musí tvořit nepřetržitý proud.

### MDIO

Toto dvou-signálové (datový a hodinový signál) rozhraní slouží primárně k přístupu do registrů připojených zařízení za účelem konfigurace (zde velmi nízká rychlost přenosu nevadí). V kontextu Ethernet rozhraní může být využíváno pro řízení externího PHY čipu.

Přístup k registru podřízeného zařízení sestává ze 16 kontrolních a 16 datových bitů. Kontrolní část je definována jako START znak následovaný příkazem (čtení/zápis), adresou zařízení a adresou registru[7]. Pokud jde o čtecí operaci, dochází po přenosu kontrolních bitů k přepnutí MDIO řadiče do stavu naslouchání – datový vodič je sdílen a využíván pro oba směry přenosu.

Rozhraní je volitelnou součástí MAC – je k dispozici MDIO řadič včetně logiky pro generování standardního 2,5 MHz hodinového signálu. Pokud je použita konfigurace MAC s podporou MDIO, je na AXI Lite zpřístupněn řídící 32 bitový registr. Do něj pak MDIO ovladač vkládá přenosové operace a čte z něj jejich výsledky.

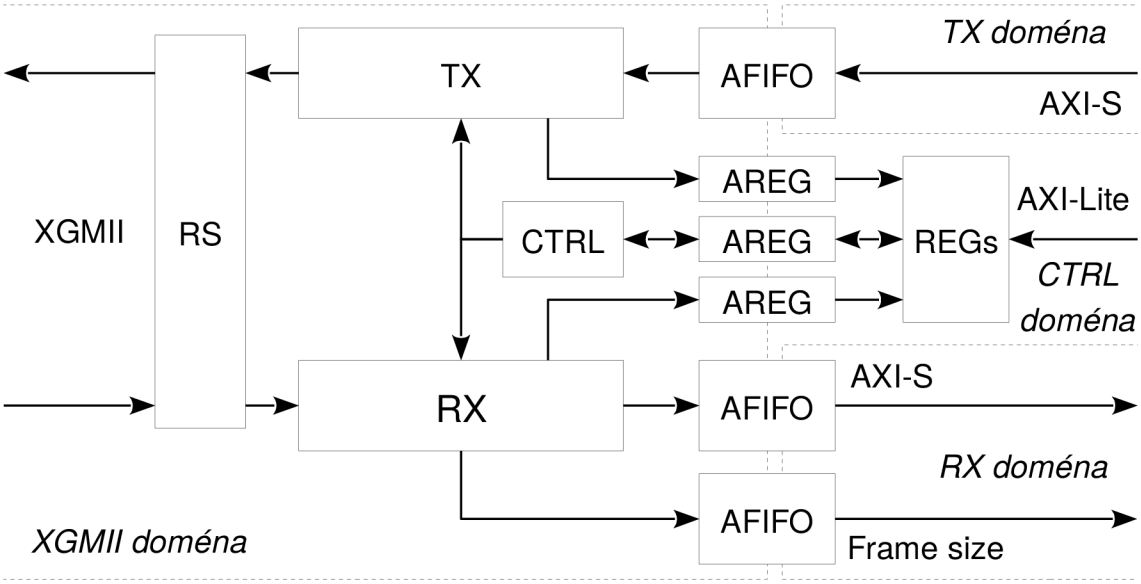
## 5.2 MAC vrstva

Implementace 10 GbE MAC navazuje na úspěšnou implementaci 1 GbE MAC pro platformu Xilinx Zynq-7000, na které jsem pracoval spolu s Filipem Brázdou v rámci předmětu projektová praxe. Některé komponenty jsou přímo použity, jiné přepracovány pro potřeby desetinásobné propustnosti. Nové komponenty musely být vytvořeny pro práci s XGMII rozhraním (1 GbE využívá odlišné RGMII).

Pro maximální jednoduchost je použita interní 64 bitová sběrnice podobná AXI-Stream. Jediný rozdíl je přidání signálu SOF (Start of Frame), který signalizuje začátek rámce. Pro dosažení plné propustnosti, jak je definována standardem pro 10 Gb Ethernet, musí toto rozhraní běžet na frekvenci 156,25 MHz – tedy stejné jako u rozhraní XGMII. V implementaci proto hlavní část MAC sdílí synchronizační signál s RS a XGMII rozhraním. Toto rozhodnutí významně zjednodušuje RS část (není třeba implementovat asynchronní přechod). Hodinový signál XGMII je generován v PHY.

### Asynchronní přechody

Asynchronní přechody vznikají na rozhraní dvou hodinových domén[10] a je bezpodmínečně nutné zajistit jejich korektní konverzi, jinak hrozí ztráta nebo poškození přenášených dat.




Obrázek 5.2: Hodinové domény MAC

V implementaci MAC se nachází 4 hodinové domény – jedna pro každé samostatné rozhraní. Jejich konkrétní podobu je možné vidět na obrázku 5.2. Tato architektura umožňuje použití odlišného hodinového signálu pro každé AXI rozhraní.

Implementace komponenty AFIFO (asynchronní FIFO) byla převzata ze serveru opencores.org. AREG (asynchronní registr) je používán pro inkrementaci čítačů (použita dvojitá synchronizace pomocí FF[11]) a pro přenos konfigurace mezi registrem a jednotkou CTRL (two-deep asynchronní FIFO).

[10] skupina komponent sdílející stejný hodinový signál
[11] Flip-Flop – klopný obvod

### Výpočet FCS

Výpočet FCS je prováděn na 64 bitech a je komplikován faktem, že poslední slovo rámce může být využito pouze částečně (využití je definováno hodnotou signálu TKEEP) a poslední krok výpočtu musí tedy proběhnout pouze na platných datech. Řešení se nabízí dvojí – použít osm paralelních CRC32 jednotek různé šířky (8, 16, 32, ... bitů) nebo zapojit kaskádově osm CRC32 jednotek šířky 8 bitů (výpočet tedy probíhá po jednotlivých bajtech a jednotka vždy čeká na předchozí).

Při návrhu bylo z důvodu významně nižší náročnosti na zdroje FPGA zvoleno druhé řešení. Výstup osmé CRC32 jednotky je tedy výsledek výpočtu pro příchozí 8bajtové slovo. V posledním slově probíhá výběr (multiplexorem s osmi vstupy) správné CRC32 jednotky na základě signálu TKEEP. Nevýhodou tohoto řešení je dlouhá kombinační cesta a tedy omezení maximální frekvence obvodu. Dosažená frekvence je ale dle výsledků syntézy nástrojem Quartus dostatečných 161 MHz. 8 bitová CRC32 jednotka je s úpravami převzatá ze serveru sigmatone.com.

### Optimalizace výkonu a zdrojů

Při implementaci synchronních obvodů s požadavkem na určitou frekvenci hodinového signálu může dojít k situaci, že syntézní nástroj není schopen splnit požadavky na časování a maximální dosažitelná frekvence je tak omezena. Nejčastější příčinou je příliš dlouhá kombinační cesta[12]. Prvním krokem při řešení tohoto problému je maximální zjednodušení kombinační logiky při zachování požadované funkcionality.

Pokud není možné problém vyřešit tímto způsobem, je třeba využít tzv. pipelining[13] a rozdělit kombinační cestu na dvě vložením registru. Při rozdělování dlouhé kombinační cesty je třeba identifikovat samostatné kroky výpočtu ideálně tak, aby byla jejich doba vykonávání podobná. Po rozdělení se zvýší maximální dosažitelná frekvence obvodu v závislosti na délce nové nejdelší kombinační cesty.

Kromě snahy o dosažení maximálního výkonu je rovněž zásadní snaha o minimalizaci množství využitých zdrojů. Významné úspory obecných zdrojů (na platformě Arria 10 zejména ALM[14]) lze dosáhnout vhodným použitím ostatních zdrojů. V rámci implementace MAC jde primárně o veliké bloky paměti využívané pro vstupní a výstupní FIFO.

**Arria 10 nabízí pro implementaci pamětí tyto prostředky[8]:**

- **LUT** (Look-Up Table, vyhledávací tabulka) – jeden ze základních stavebních bloků FGPA, využíváno pro implementaci mnoha logických operací
- **MLAB** (Memory Logic Array Block) – malé bloky paměti v rámci ALM, vhodné pro posuvné registry a široké FIFO s nízkou hloubkou
- **M20K blok** – dedikované bloky o velikosti 20 Kb, vhodné pro velká pole paměti, nevýhodou jsou jistá technická omezení přístupu do této paměti

Při implementaci MAC jsem se soustředil na obě oblasti. Pipelining byl nutný v TX části, kde FCS generátor nedosahoval dostatečného výkonu – vložením registrů okamžitě před a za CRC32 kaskádu byl problém vyřešen. Pro implementaci velkých pamětí v MAC byly použity M20K bloky a omezena tak spotřeba ostatních zdrojů.

[12] logika a propoje, kterými signál prochází na cestě mezi dvěma registry
[13] zřetězené zpracování
[14] Adaptive Logic Module

## 5.3 DPDK rozhraní

Jako základ DPDK ovladače pro vytvořené 10 Gb Ethernet rozhraní byl použit ovladač `sring` pro DMA vytvořený Bc. Michalem Orsákem. Jeho primárním úkolem je inicializace celého rozhraní (tedy MAC i DMA) a jeho zpřístupnění DPDK aplikacím (pro ověření funkčnosti byla použita dostupná `sring_test_app`). Dále musí poskytovat rozhraní pro změnu konfigurace a získávání statistik provozu[3].

### Inicializace DPDK rozhraní

Pro každé DPDK rozhraní je nutné vytvořit a zaregistrovat strukturu `rte_vdev_driver`, která obsahuje adresy funkcí `probe` a `remove`. Probe[15] je volána při inicializaci EAL a vytváří rozhraní. Remove je analogicky volána při uzavírání rozhraní a provádí zastavení všech komponent a jejich uvedení do výchozího stavu.

Při vytváření rozhraní je nejprve alokována paměť pro uložení interních dat ovladače, která jsou nutná pro uchovávání stavu. Zahrnují mimo jiné tyto položky:

- fronta deskriptorů pro DMA
- rte_mempool pro alokaci nových mbufů
- buffer pro uložení délek přijímaných paketů
- MAC adresu rozhraní
- fyzické adresy DMA řadiče a MAC modulu použitelné pro přístup k jejich registrům

Dalším krokem je mapování virtuálních adres DMA a MAC na adresy fyzické. Ty musí být nastaveny v syntézním nástroji a také v konfiguraci DPDK ovladače před jeho překladem. Samotné mapování je prováděno standardní funkcí `mmap`. Při úspěchu je ještě čtením stavového registru provedena kontrola identity namapované komponenty. Pokud jedna z těchto operací skončí neúspěchem, DPDK aplikace vypíše chybu a je ukončena.

Po inicializaci HW komponent probíhá příprava potřebných DPDK struktur – zejména `rte_eth_dev`, `rte_eth_dev_data` a fronty mbufů pro RX a TX. Ty jsou naplněny vhodnou výchozí konfigurací, MAC adresami rozhraní a jsou jim přiřazeny funkce pro řízení (struktura `dev_ops`) a příjem/odesílání dávky paketů (`rx_pkt_burst` a `tx_pkt_burst`).

Protože jeden DMA řadič může obsluhovat více Ethernet rozhraní, musí být ovladač schopen inicializovat a ovládat obecný počet MAC modulů.

### Operace Ethernet rozhraní

Ovladač poskytuje množinu funkcí pro řízení DPDK rozhraní, zejména start (spuštění), stop (zastavení), přidání/odstranění MAC adresy a získávání statistik či informací o konfiguraci rozhraní.

Při spuštění rozhraní je vykonán restart připojených MAC modulů a nulování jejich statistických čítačů. Při zastavení jsou korektně zastaveny všechny TX i RX fronty a MAC moduly vypnuty. Funkce `eth_dev_info` poskytuje informace o vlastnostech rozhraní – maximální počet MAC adres (závisí na konfiguraci MAC modulu, teoretické maximum je 7) a maximální délka příchozího paketu (dáno velikostí vstupní fronty zvolené při syntéze). Funkce pro získání statistik zajišťuje atomický přístup k registrům MAC. K dispozici je také funkce pro jejich vynulování.

[15] prohledat

# Kapitola 6

# Testování a výsledky

Testování HW komponent bylo prováděno ve třech fázích:

1. **VHDL**[1] **testbench** – jednoduché, úzce zaměřené testy jednotlivých komponent
2. **Funkcionální verifikace** – UVM[2] model v jazyce System Verilog
3. **Ověření implementace na cílové platformě**

Simulace prvních dvou fází byly spouštěny v programu Modelsim od společnosti Mentor. Pro následné ladění a sledování signálů ve třetí fázi byl použit Intel SignalTap II. Zachytávání síťového provozu a práci se soubory typu PCAP umožnil program Wireshark. Testování SW probíhalo experimentálně s využitím ladících režimů a výpisů na platformě, která byla k dispozici.

## 6.1 VHDL testbench

Při vývoji HW je velice důležité průběžné ověřování funkčnosti jednotlivých komponent a později i větších celků. Ideálně je pro každou VHDL entitu vytvořen tzv. VHDL testbench (testovací prostředí), který generuje vstupy (stimulus) pro instanci VHDL entity a následně kontroluje její výstupy. Tyto testy jsou pak spouštěny v simulačním SW (například Modelsim), který je schopen analyzovat VHDL a modelovat jeho chování v reálném čipu. Je ale nutné brát na vědomí, že VHDL simulátory nejsou bezchybné a že některé VHDL konstrukce není možné syntetizovat. Implementace, která je plně funkční v simulacích, tedy nemusí stejným způsobem fungovat na reálné platformě.

Složitost testů se liší dle potřeb vývoje a může se jednat například o jednoduché, ručně definované vstupy spolu s ruční kontrolou výstupů, jak jsou zaznamenány simulátorem. Tyto testy jsem využíval ke kontrole správnosti VHDL kódu během samotného vývoje jednotlivých komponent. Dalším krokem jsou tzv. regresní testy – tyto typicky automaticky kontrolují výstupy entit a jsou používány pro opětovné testování již funkčního chování obvodu při jeho modifikacích (například optimalizaci výkonu či spotřeby zdrojů). Složitější testy jsou pak schopny generovat náhodný stimulus a kontrolovat správnost výsledků – pro tyto účely se ale používá téměř výhradně funkcionální verifikace a jazyk System Verilog (viz kapitola 6.2).

[1] VHSIC Hardware Description Language
[2] Universal Verification Methodology

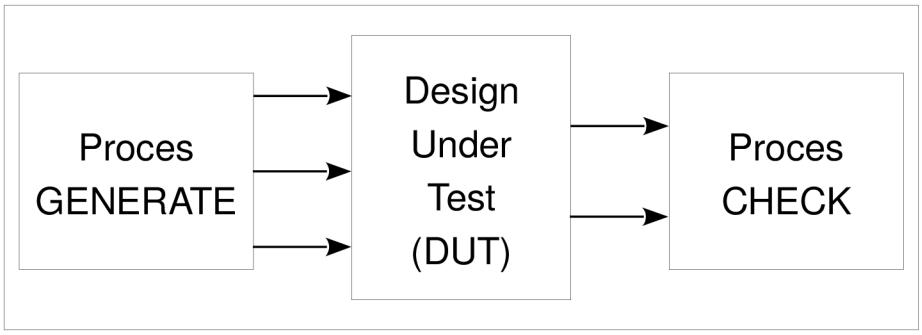
Jazyk VHDL byl původně vytvořen pro dokumentaci chování ASIC čipů dodávaných Ministerstvu obrany USA a následně pro jejich simulaci za použití VHDL simulátorů[16]. Obsahuje proto širokou paletu nástrojů a jazykových konstrukcí, které nemají v číslicových obvodech ekvivalent a není je tedy možné syntetizovat. Tyto konstrukce (mnohdy implementující vysoko-úrovňové chování známé z programovacích jazyků) je ale možné bez jakéhokoliv omezení využít při tvorbě VHDL testů. Díky tomu je často možné velmi jednoduše a rychle implementovat chování relativně složité komponenty. Testování pak probíhá porovnáváním výstupů modelu a VHDL entity.

### Prostředky VHDL pro simulaci a testování

- **práce se soubory** – toto je možné využít zejména pro načítání reálného vzorku vstupních dat nebo generování vstupů jiným nástrojem
- **ladící výpisy**
- **aserce** (angl. assertions) – slouží pro kontrolu hodnot signálů a případné vyvolání chyby nebo varování v průběhu simulace
- **příkaz wait** – umožňuje čekat přesný časový úsek nebo na splnění určitých podmínek
- **generování náhodných čísel**
- **více-hodnotové logické signály** – oproti dvou-hodnotovým signálům nesou další informace (například U – nedefinováno nebo Z – vysoká impedance)

### Architektura testovacího prostředí

VHDL testbench obvykle sestává ze tří částí – generování vstupů (pro korektní simulaci musí být definovány všechny vstupy), kontroly výstupů a instance testované komponenty (DUT, také nazýváno UUT[3] nebo DUV[4]). Entita, ve které jsou tyto části implementovány se typicky nazývá `Top level` (angl. nejvyšší úroveň) a je spouštěna ve VHDL simulátoru. Před spuštěním simulace je ještě nutné definovat dobu jejího běhu (simulaci lze ale předčasně přerušit nebo její běh prodloužit) a signály, jejichž průběh má být zaznamenáván. Výstupem je pak tzv. waveform soubor, kde je možné prohlížet změny signálů v čase.




Obrázek 6.1: Architektura VHDL testbench

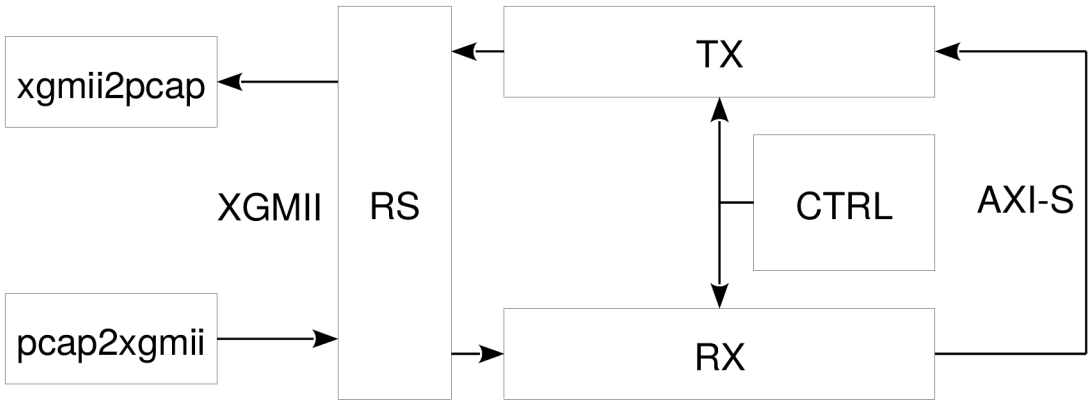
[3] Unit Under Test
[4] Design Under Verification

### MAC loopback test

V rámci práce jsem vytvořil tzv. loopback[5] test kompletního návrhu MAC, kde je výstup RX části (rozhraní AXI-Stream) použit jako vstup TX části. V promiskuitním režimu by pak měl výstup TX odpovídat vstupu RX (rozhraní XGMII). Tento způsob testování umožňuje poměrně jednoduchou kontrolu funkčnosti celého systému.

V prvním kroku jsem definoval jednoduchý XGMII stimulus a sledoval průběh signálů mezi jednotlivými komponentami MAC. Při chybném výstupu některé z komponent je možné sledovat stavy jednotlivých signálů, což výrazně napomáhá při odhalování chyb v návrhu nebo implementaci.

Pro urychlení a automatizaci testování jsem následně vytvořil komponenty `pcap2xgmii` a `xgmii2pcap`, které jsou schopny převádět standardní PCAP[6] soubor na XGMII stimulus a naopak. Po spuštění simulace pak stačí porovnat obsah vstupního a výstupního PCAP souboru – pokud se shodují, MAC funguje pro daný vstup správně. Porovnávání PCAP souborů bylo realizováno skrze volně šiřitelný skript `pcap_diff` využívající Python knihovnu `scapy`. Pro zajištění co největší rozmanitosti vstupů (a tedy odhalení chyb v okrajových případech) jsem použil soubory PCAP s různými druhy paketů, většinou zachycenými z reálného síťového provozu.




Obrázek 6.2: Loopback VHDL testbench

### Další testy

Několika dalšími testy byla ověřena funkčnost RX výstupního rozhraní s délkami přijatých rámců – délka paketů na AXI-Stream rozhraní byla v testu spočtena a porovnána s délkou, kterou dal k dispozici MAC. Otestovány byly také funkce řídící jednotky, konkrétně spouštění s různou konfigurací, změna konfigurace při běhu a korektní funkce statistických čítačů a přístupu k nim.

V RX části byla rovněž otestována správná detekce chyb na XGMII rozhraní, korektní výpočet FCS a funkce filtru MAC adres – v režimu filtrování i v promiskuitním režimu. Během testování bylo odhaleno několik chyb v návrhu a implementaci – zejména při zřetězeném výpočtu CRC-32 a zajištění korektní mezery dle DIC.

[5] zpětná smyčka – výstup RX je použit jako vstup TX

[6] soubor pro ukládání síťového provozu zachyceného knihovnou libpcap (pcap – packet capture)

## 6.2 Funkcionální verifikace

Cílem funkcionální verifikace je ověřit korektnost chování DUT vkládáním náhodně generovaných vstupních dat a kontrolou výstupu. Základním předpokladem úspěšné verifikace je pokrytí všech možných vstupů ve všech platných stavech DUT. Během verifikace pak typicky probíhá kontrola, zda DUT je stále v platném stavu a zda jeho výstupní data odpovídají očekávání. Často je také sledováno pokrytí kódu – podmínek, výrazů a stavů FSM. Pokud jsou některé části kódu nepokryté, znamená to, že vstupní data nepokrývají všechny možnosti, nebo že jde o tzv. mrtvý kód (případně chybně vytvořená VHDL konstrukce, kterou simulátor vyložil jinak, než vývojář zamýšlel). Tyto funkce poskytují VHDL simulátory určené pro verifikaci (například Modelsim nebo novější Questa).

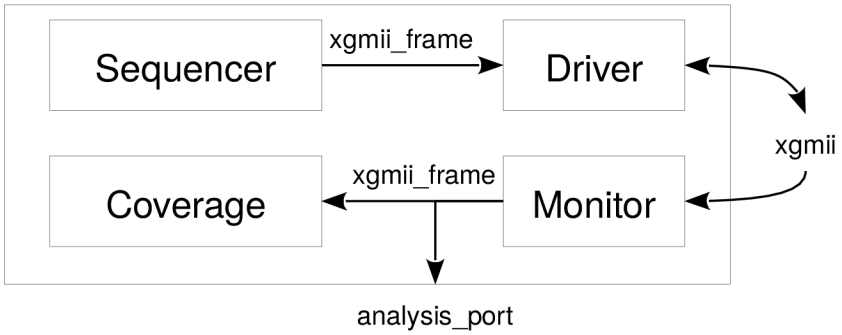
### System Verilog

VHDL postrádá některé moderní a vysoko-úrovňové vlastnosti (zejména OOP) a není příliš vhodné pro tvorbu komplexního verifikačního prostředí. Proto byl vytvořen jazyk System Verilog, který obohacuje původní Verilog o OOP, některé chybějící vlastnosti z VHDL a schopnosti jazyků pro verifikaci HW (např. Vera). Zároveň cílí na rychlou tvorbu modelů (poskytuje například dynamická pole, fronty a další struktury) zatímco VHDL provádí velké množství kontrol při překladu, má poměrně striktní syntax, je silně typované a cílí spíše na pomalejší vývoj a zachycení maximálního množství chyb na začátku vývojového procesu[16]. V neposlední řadě je implementován kompletní systém asercí SVA[7] a DPI (Direct Programming Interface), který umožňuje přímé volání C funkcí.[6]

### UVM

UVM (Universal Verification Methodology) je knihovna tříd implementovaná v jazyce System Verilog. Jejím cílem je poskytnout nástroje a prostředí umožňující tvorbu znovupoužitelných transakčních modelů pro funkcionální verifikaci HW komponent. Při správném použití UVM je možné vytvářet velmi rozdílné verifikační prostředí (UVM Testbench) při minimálních změnách – jednotlivé komponenty je možné dle potřeby jednoduše upravit využitím třídní dědičnosti[1].

Jednou z nejdůležitějších komponent je Agent, který je vytvářen pro každé použité rozhraní (v této práci např. XGMII Agent). Jeho primárním úkolem je obsluhovat konkrétní rozhraní, přičemž stejný Agent může operovat ve výstupním i vstupním režimu.



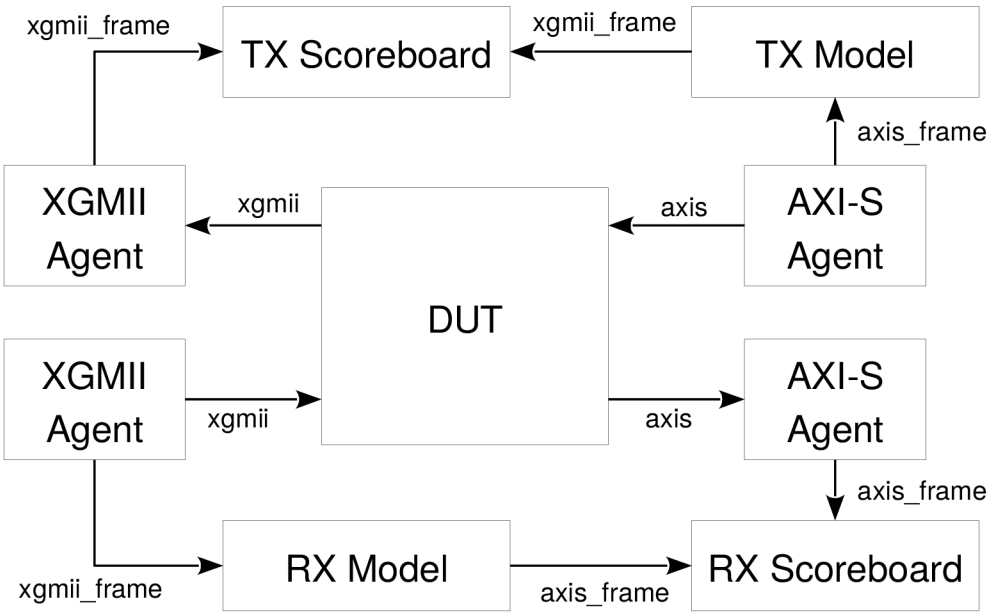

Obrázek 6.3: Schéma XGMII Agenta

[7] SystemVerilog Assertions

Sequencer generuje rámce na základě nastavení v `uvm_test` a odesílá je komponentě Driver, která má za úkol řídit samotné rozhraní (dle potřeby v režimu master nebo slave). Aktivitu na rozhraní sleduje Monitor a znovu vytváří rámce, které jsou využity pro sledování pokrytí vstupů/výstupů a případně odeslány do komponenty Scoreboard.

### Model pro testování MAC

Na rozdíl od VHDL loopback testu probíhá verifikace nezávisle pro RX a TX části MAC. Důvodem je poměrně nízká přidaná složitost implementace (opačný směr je tvořen převážně stejnými komponentami) a také snaha o detekci dvojitých chyb – teoreticky je možné, aby v RX i TX nastala specifická chyba a jejich kombinace vedla ke zdánlivě správnému výsledku.




Obrázek 6.4: Schéma UVM modelu MAC

Jednotlivé části modelu jsou poměrně jednoduché a mají striktně oddělené role. Scoreboard tak například nemusí analyzovat XGMII a AXI-Stream, ale jen porovnává dva objekty stejné třídy (`xgmii_frame`). Sledování pokrytí stimulu probíhá v příslušném agentu na vstupní straně a verifikace pokračuje, dokud není dosaženo 100% pokrytí – jedním kritériem může být například odeslání alespoň jednoho paketu pro každou podporovanou délku.

Samotná data náhodně generuje XGMII Agent připojený na vstupní XGMII rozhraní. RX Model pak provádí dle příslušných specifikací transformaci `xgmii_frame` na `axis_frame`. Výsledek této transformace by měl odpovídat výstupu RX části MAC zachyceného agentem pro AXI-Stream (opak značí chybu v implementaci). Verifikace TX probíhá analogicky.

Verifikace je spouštěna a řízena třídou `test`, přičemž je možné spouštět více různých testů nad stejným UVM modelem. Průběh verifikace zaznamenávají funkce pro zápis do logů (s nastavitelnou úrovní citlivosti – při vývoji UVM modelu budeme vypisovat zprávy pro ladění s citlivostí DEBUG). Verifikace je typicky přerušena při výskytu chyby. Můžeme ale stanovit, že v takovém případě bude pokračovat dále – takový režim může být žádaný, pokud chceme zaznamenat chování testované komponenty po výskytu chyby.

## 6.3 Ověření implementace na cílové platformě

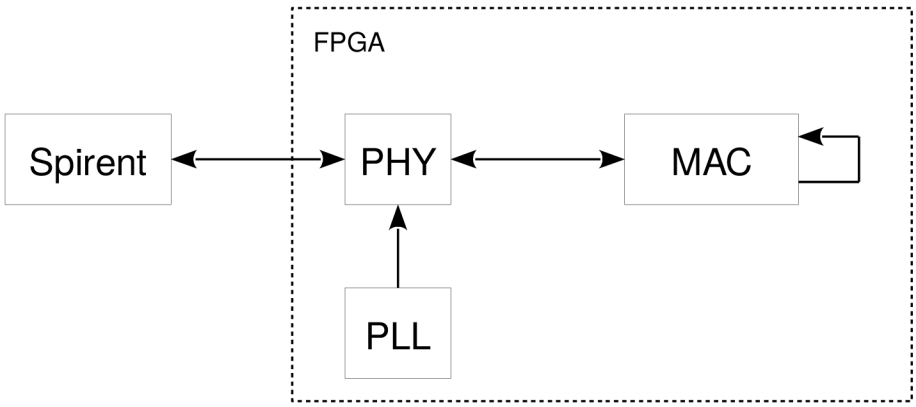
Poslední fáze testování spočívá ve vytvoření Quartus projektu pro konkrétní platformu, syntéze samotného obvodu a jeho naprogramování do FPGA čipu dostupného v rámci platformy Arria 10 SoC.

Jako testovací platforma byla použita deska pro vývojáře Reflex CES Achilles Arria 10 SoC SoM s PCIe deskou, která nese dva SFP+ sloty a jeden QSFP+[8] slot. Konfigurace a programování FPGA čipu probíhala přes standardní rozhraní UART. Pro spouštění a zavádění OS byl použit systém Buildroot, který pro urychlení testování načítal konfiguraci a rootfs Linuxu z TFTP serveru (nebylo tedy nutné při každé změně vysunout paměťovou SD kartu a nakopírovat na ni novou konfiguraci).

Při testování pak byl využit nástroj SignalTap, který umožňuje před syntézou určit, které signály mají být sledovány a kdy mají být jejich hodnoty zaznamenány. To určuje tzv. trigger (spoušť) – obvykle je nastavována hodnota nebo množina hodnot, jejichž výskyt spustí záznam. Zaznamenaná data je pak možné zobrazit podobně jako výsledky VHDL simulace. Zásadním problémem je ale nutnost při každé změně spouštět syntézu znova (to trvá v našem případě přibližně 15 minut) a je proto vhodné provádět důkladné testování a verifikace v dřívějších fázích vývoje.

### Testování MAC

Testování MAC probíhalo v různých konfiguracích, ale primárně v režimu looback a to jak v rámci jednoho Ethernet rozhraní (RX –> TX), tak s využitím dvou samostatných rozhraní (RX0 –> TX1, RX1 –> TX0). Po odstranění všech problémů byl vytvořen testovací projekt v rámci projektu SProbe, kde bylo využito všech šest 10GbE Ethernet rozhraní v reálném provozu. Pro generování vstupů a analýzu výstupů byl nejprve využit osobní počítač s 10 GbE síťovou kartou, programem Wireshark a PCAP soubory. V druhé fázi testování byl namísto toho použito samostatné zařízení společnosti Spirent schopné generovat dostatek paketů pro plné vytížení 10 GbE linky.



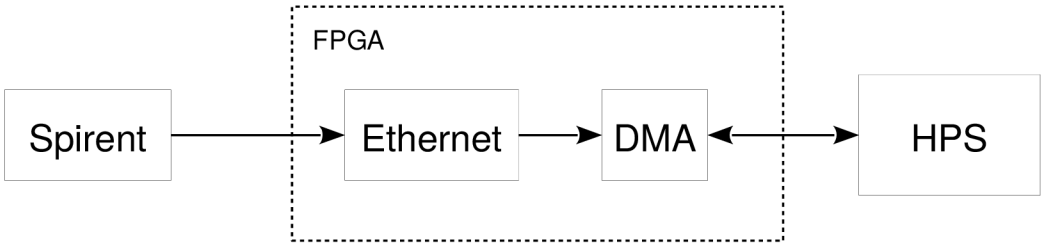

Obrázek 6.5: Schéma testování MAC

[8] Quad SFP+ – používáno pro 40 GbE v konfiguraci 4x10 GbE

Pro jednoduchost použití MAC v dalších projektech jsem vytvořil definici IP core pro Qsys[9] – soubor `fitmac_10g_hw.tcl`. Uživatel tak vůbec nemusí zasahovat do souborů MAC nebo ručně vytvářet instanci VHDL entity.

### Testování DMA a DPDK

Po úspěšném otestování MAC byl DMA řadič přidán do Quartus projektu a připojen k HPS . Testování pak probíhalo zasíláním paketů na Ethernet rozhraní a spuštěním testovací DPDK aplikace, jejíž úkolem bylo řídit DMA řadič a příchozí pakety přijmout (pro kontrolu byl vypisován jejich obsah).




Obrázek 6.6: Schéma testování DMA

Během této fáze testování jsem narazil na množství problémů plynoucích z nepřesné nebo nejasné dokumentace pro Arria 10 SoC a ARM Cortex-A9. Bylo také nutné provést několik úprav samotného DMA řadiče, aby jej bylo možné přeložit pro Arria 10 (problémy byly zejména s odlišnostmi v pravidlech pro použitá rozhraní).

Po vyřešení všech těchto problémů je DMA řadič plně funkční. Zůstává ovšem problém při využití AXI3 s podporou ACP, které z neznámých důvodů nefunguje dle očekávání (dochází ke korupci dat při čtení i zápisu) navzdory jeho použití přesně dle specifikace a doporučení společností Intel a ARM. Ve snaze zjistit příčinu tohoto problému byla do DMA doplněna řada registrů pro manuální konfiguraci AXI sběrnice za běhu (zejména signálů AxCACHE a AxUSER) a byly zkoušeny i jiné než doporučené konfigurace. Tento proces bohužel problém nevyřešil, ale alespoň bylo dokázáno, že ACP skutečně nějakým způsobem aktivní je – chování AXI se měnilo. Při vypnutí ACP se sběrnice chová korektně.

Pokusy o zprovoznění ACP (a tedy zvýšení výkonu) skončily v době psaní této práce změnami konfigurace OS Linux a porovnáváním nastavení této platformy s původní platformou DMA řadiče Xilinx Zynq, které rovněž používá jádra ARM Cortex-A9 a rozhraní ACP. Bylo aktivováno několik relevantních Errata workaround[10] a dokonce experimentálně modifikován zdrojový kód některých funkcí OS Linux – bohužel bez kladného výsledku.

[9] program pro grafické spojování a nastavovaní jednotlivých IP core; výstup je možné použít v Quartus projektu

[10] úprava chování OS pro softwarovou opravu (doslova obejití) chyby nalezené v HW

## 6.4 Výsledky

Úspěšně byl implementován a otestován 10GbE MAC pro platformu Arria 10. V současnosti se již používá pro testování nové platformy v rámci projektu SProbe. Dále byl na platformu Arria 10 úspěšně portován a otestován dostupný DMA řadič. Zde jsem ale bohužel narazil na problémy při použití rozhraní AXI s podporou ACP, kdy náhodně dochází ke čtení chybných dat a je tedy v současnosti nutné pro ukládání meta informací alokovat nekoherentní paměť. To způsobuje propad výkonu DMA řadiče oproti původní platformě Zedboard (čip Xilinx Zynq) a znemožňuje měření reálného výkonu. Důvod problému je momentálně neznámý (ACP je v rámci DMA používáno dle specifikace a dokumentace platformy Arria 10) a na jeho řešení pracuji. Samotný MAC je schopen zpracovávat plnou propustnost 10 Gb Ethernetu (měřeno s využitím zařízení Spirent).

### Spotřeba zdrojů

V následujících tabulkách je přehled spotřebovaných FGPA zdrojů projektu s jedním MAC, PHY a dalšími potřebnými komponentami (bez DMA). Překlad probíhal v Intel Quartus Prime 17.0 Standard.

Tabulka 6.1: RX + TX

| **Typ zdroje** | **Počet** |
|---|---|
| ALMs | 2323 |
| ALUT | 2988 |
| Dedicated logic registers | 2096 |
| Block memory bits | 160484 |

Tabulka 6.2: Pouze RX část

| **Typ zdroje** | **Počet** |
|---|---|
| ALMs | 1377 |
| ALUT | 1591 |
| Dedicated logic registers | 1517 |
| Block memory bits | 87716 |

Tabulka 6.3: Pouze TX část

| **Typ zdroje** | **Počet** |
|---|---|
| ALMs | 1222 |
| ALUT | 1357 |
| Dedicated logic registers | 631 |
| Block memory bits | 80448 |

Vytvořený SW byl úspěšně zakomponován do systému Buildroot, který je používán pro překlad OS a tvorbu boot image pro využívanou platformu. Existující DPDK ovladač pro použitý DMA řadič byl rozšířen o ovladač pro MAC a otestován pomocí jednoduché testovací aplikace. Také byl vytvořen jaderný ovladač pro OS Linux. Ten je využíván pro konfiguraci MAC na platformách bez DPDK, kde není vyžadována výměna paketů mezi MAC a OS (pakety jsou dále zpracovávány v FPGA).

### Měření propustnosti

Kvůli problémům s nefunkčností ACP rozhraní bylo nutné pracovat přímo s hlavní pamětí bez využití cache procesoru. To zásadním způsobem omezuje výkon zejména pro krátké pakety, kde režie tvoří významnou část provozu.

Tabulka 6.4: Propustnost Ethernet rozhraní s vypnutým cache systémem

| **Délka paketů** | **Mpkt/s** | **Mb/s** |
|---|---|---|
| 64B | 1,712 | 876,544 |
| 65B | 1,709 | 888,68 |
| 128B | 1,691 | 1731,584 |
| 256B | 1,648 | 3375,104 |
| 512B | 1,548 | 6340,608 |
| 1500B | 0,748 | 8976 |

Měření odpovídají výsledkům s vypnutým ACP na čipu Xilinx Zynq-7000, který obsahuje stejné CPU. Je zřejmé, že výkon je limitován přístupy do RAM bez vyrovnávací paměti a to jak ze strany CPU, tak DMA řadiče (při použití ACP by přistupoval přímo do L2 cache a nebylo by tedy nutné provádět dva přístupy do hlavní paměti navíc).

Za povšimnutí stojí měření propustnosti pro pakety o velikost 64 a 65 bajtů – pakety větší než 64B je totiž nutné rozdělit do dvou transakcí. To by za běžných okolností vedlo k propadu výkonu. Použité DMA ale podporuje prokládání transakcí, které tento problém zakrývá a výkon tak zůstává stabilní.

Po zprovoznění ACP lze očekávat řádově vyšší výkon (zejména při přenosech malých paketů). Další nárůst výkonu oproti původní platformě přinese o 56% rychlejší takt všech AXI sběrnic. Pro běžný provoz (kombinace menších a větších paketů) tedy lze očekávat dosažení plné 10 Gb/s propustnosti.

# Kapitola 7

# Závěr

Cílem této práce bylo navrhnout, implementovat a otestovat 10 Gb Ethernet rozhraní pro platformu Arria 10 SoC. Tento cíl byl splněn a výsledek je již v současnosti reálně používán v několika systémech vytvořených na FIT. Druhotným cílem práce byl návrh a implementace DPDK ovladače pro vytvořené Ethernet rozhraní a změření jeho vlastností. Ovladač pro zpřístupnění rozhraní DPDK aplikací byl úspěšně vytvořen a otestován. Měření bylo provedeno pouze pro DPDK aplikaci bez využití vyrovnávací paměti CPU.

Při implementaci MAC se podařilo dosáhnou velice nízké spotřeby zdrojů FPGA. Komponenta pak byla důkladně otestována v rámci VHDL simulací, funkcionální verifikace s využitím knihovny UVM a také v reálném provozu na cílové platformě. Tuto komponentu je možné použít v libovolných projektech vyžadujících 10 GbE konektivitu pro FPGA čip. V současnosti je podporována pouze platforma Arria 10, která je využita v rámci platformy ARMINDA (produkt projektu SProbe). S minimální snahou je ale možné podporovat výkonnější platformu stejné firmy Stratix 10 a po jistých úpravách také platformu UltraScale od firmy Xilinx.

Práce se dále zabývala využitím DPDK namísto běžného síťového zásobníku v OS Linux. DPDK ovladač pro konfiguraci, příjem a odesílání paketů byl úspěšně vytvořen a celé 10 Gb Ethernet rozhraní je tedy možné plnohodnotně používat. Pro potřeby konfigurace v projektech, které nevyužívají DPDK byl implementován ovladač použitelný v rámci síťového zásobníku OS Linux. Ten nepodporuje odesílání a přijímání paketů, ale zpřístupňuje statistické informace a umožňuje úplnou konfiguraci včetně nastavování filtru MAC adres.

V rámci práce rovněž proběhla portace existujícího DMA řadiče pro platformu Xilinx Zynq-7000 na platformu Intel Arria 10 Soc. DMA je plně funkční, ale jeho výkon je v současnosti bohužel značně omezen nefunkčností koherentního přístupu do cache systému (rozhraní ACP) – při použití ACP došlo na originální platformě k více než dvojnásobnému zrychlení. Není tedy možné využít vyrovnávací paměť CPU, což zásadním způsobem omezuje výkon celého rozhraní.

Na výsledky této práce je možné navázat implementací dalších funkcí a pokračováním práce na ovladači pro OS Linux (pak by bylo možné použít implementované 10 GbE rozhraní jako plnohodnotnou síťovou kartu pro běžné použití). Novými funkcemi mohou být například architektura cut-through pro FIFO v MAC pro snížení latence odesílaných rámců nebo přidání podpory pro další rychlostní režimy protokolu Ethernet. V současnosti pokračuje práce na zprovoznění rozhraní ACP a tedy zvýšení výkonu celého systému. Alternativou může být využití jiného DMA řadiče (MAC není závislý na konkrétní implementaci).

# Literatura


[1] Accellera: *Universal Verification Methodology (UVM) 1.2 User's Guide*. [Online; navštíveno 13.05.2018].
URL http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf

[2] Brouer, J. D.: *The calculations: 10Gbit/s wirespeed*. [Online; navštíveno 8.05.2018].
URL http://netoptimizer.blogspot.cz/2014/05/the-calculations-10gbits-wirespeed.html

[3] DPDK: *Programmer's Guide*. [Online; navštíveno 5.05.2018].
URL http://dpdk.org/doc/guides/prog_guide/

[4] DPDK: *rte_mbuf Struct Reference*. [Online; navštíveno 4.05.2018].
URL http://dpdk.org/doc/api/structrte__mbuf.html

[5] Foundation, T. L.: *networking:napi*. [Online; navštíveno 5.05.2018].
URL https://wiki.linuxfoundation.org/networking/napi

[6] IEEE: *1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. [Online; navštíveno 13.05.2018].
URL https://ieeexplore.ieee.org/document/8299595/

[7] IEEE: *802.3-2015 - IEEE Standard for Ethernet*. [Online; navštíveno 8.05.2018].
URL https://ieeexplore.ieee.org/servlet/opac?punumber=7428774

[8] Intel: *Intel Arria 10 Core Fabric and General Purpose I/Os Handbook*. [Online; navštíveno 12.05.2018].
URL https://www.altera.com/documentation/sam1403483633377.html

[9] Intel: *Intel Arria 10 Hard Processor System Technical Reference Manual*. [Online; navštíveno 11.05.2018].
URL https://www.altera.com/documentation/sfo1410070178831.html

[10] Intel: *Intel Arria 10 Transceiver PHY User Guide*. [Online; navštíveno 12.05.2018].
URL https://www.altera.com/documentation/nik1398707230472.html

[11] ITU: *X.200 : Information technology - Open Systems Interconnection - Basic Reference Model: The basic model*. [Online; navštíveno 8.05.2018].
URL http://www.itu.int/rec/T-REC-X.200-199407-I/en

[12] Limited, A.: *AMBA 4 AXI4-Stream Protocol*. [Online; navštíveno 12.05.2018].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html

[13] Limited, A.: *AMBA AXI and ACE Protocol Specification.* [Online; navštíveno 12.05.2018].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022f.b/index.html

[14] Lynskey, E.: *10 Gigabit Ethernet Deficit Idle Count Tutorial.* [Online; navštíveno 10.05.2018].
URL https://www.iol.unh.edu/sites/default/files/knowledgebase/10gec/10GbE_DIC.pdf

[15] Rosen, R.: *Linux Kernel Networking: Implementation and Theory.* Apress, 2014, ISBN 978-1-4302-6196-4.

[16] Rushton, A.: *VHDL for Logic Synthesis, Third Edition.* John Wiley & Sons, Ltd., 2011, ISBN 978-0-470-68847-2.

[17] Woods, R.; McAllister, J.; Lightbody, G.; aj.: *FPGA-based Implementation of Signal Processing Systems, Second Edition.* John Wiley & Sons, Ltd., 2017, ISBN 978-1-119-07795-4.