FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání diplomové práce

Název:	FPGA IP jádro pro síťové rozhraní s podporou v Linuxu
Student:	Bc. Jan Brokeš
Vedoucí:	Ing. Tomáš Beneš
Studijní program:	Informatika
Obor / specializace:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Analyzujte existující řešení síťového rozhraní v Xilinx FPGA.

Navrhněte architekturu univerzálního síťového rozhraní, které bude podporovat rychlosti 1G a 10G umožňující dynamické nebo statické přepínání mezi nimi. Síťové rozhraní musí podporovat více připojených AXI4 stream konzumentů. Navrhněte způsob řešení filtrování paketů pro jednotlivé konzumenty pomocí extrakce dat z aplikačního protokolu v příchozích paketech.

Jedním z nich musí být CPU s Linuxovým operačním systémem. Buď použijte existující ovladač v linuxovém jádře nebo navrhněte vlastní.

Všechny vámi navržené bloky otestuje v simulaci a ověřte funkčnost vámi navrženého řešení na FPGA.

Elektronicky schválil/a doc. Ing. Hana Kubátová, CSc. dne 10. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

FPGA IP jádro pro síťové rozhraní s podporou v Linuxu

Bc. Jan Brokeš

Katedra číslicového návrhu

Vedoucí práce: Ing. Tomáš Beneš

27. června 2021

Poděkování

Děkuji mému vedoucímu Ing. Tomáši Benešovi za jeho rady a vedení při tvorbě této práce. Také děkuji mým nejbližším za jejich podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 27. června 2021 ……

Odkaz na tuto práci

Brokeš, Jan. *FPGA IP jádro pro síťové rozhraní s podporou v Linuxu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Cílem práce je implementace IP jádra, které zpracovává SFP 10G nebo 1G signál na desce ZC706 do formy paketů. Ty potom filtruje podle cílového portu, určené pakety posílá na AXI Stream rozhraní v FPGA logice. Veškerý zbylý provoz předává ovladači v OS Petalinux.

Výsledné řešení z velké části funguje tímto způsobem. Přijímá data z SFP transceiveru, rozdělí je na pakety a ty filtruje na základě cílového portu, vybrané posílá do FPGA, ostatní do OS. Kvůli omezení od výrobce nebylo možné implementovat dynamické přepínání mezi rychlostmi, pouze statické. 1G verze vyžaduje specifické nastavení v OS po prvním spuštění, 10G funguje bez problému.

Klíčová slova IP jádro, Zynq, Petalinux, 1 Gbit, 10 Gbit, 1G, 10G, FPGA, Xilinx, síťová karta, síťové rozhraní.

Abstract

The goal of this thesis is the implementation of IP core for processing SFP 10G or 1G signal on ZC706 board into packets. The packets are filtered based on destination port, specified packets are sent to AXI Stream interface in FPGA logic. All other traffic is sent to network driver in OS Petalinux.

The solution largely works this way. It receives data from SFP transceiver, splits them into packets, and filters them based on destination port, chosen ones are sent to FPGA, others to OS. Because of limitation from mafunacturer, it isn't possible to implement dynamic switching, only static. 1G version requires specific configuration after boot, 10G works without an issue.

Keywords IP core, Zynq, Petalinux, 1Gbit, 10 Gbit, 1G, 10G, FPGA, Xilinx, network card, network interface.

Obsah

Seznam obrázků

Úvod

V oblasti telekomunikací je klíčové rychlé připojení k síti. V náročnějších oblastech je běžná rychlost přenosu 1 Gb/s, je tedy nutné ji podporovat. Rychlost 10 Gb/s se v dnešní době rychle rozšiřuje v sítích, které obsluhují velké objemy dat, a začíná postupně nahrazovat 1 Gb/s, proto je žádoucí podporovat i tuto rychlost.

FPGA se často využívají právě pro zpracování vysokorychlostního síťového provozu, kvůli jejich efektivitě a flexibilitě. Standardy určující dříve zmíněné rychlosti se zásadně liší, každý vyžaduje jiný FPGA design, a proto mezi nimi nelze snadno přepínat. Další problém s použitím FPGA jako síťového zařízení je náročná implementace základních protokolů. Toto řeší SoC nazvané Zynq – spojuje FPGA, které může rychle provádět náročné výpočty, s běžným CPU, na němž se snadněji implementují ostatní části aplikace, například komunikace s jinými zařízeními.

Cílem práce je vytvořit IP jádro do FPGA na desce Zynq kompatibilní s 1 Gb/s i 10 Gb/s, schopné část provozu určenou hodnotou cílového portu posílat aplikaci na FPGA, zatímco vše ostatní předá CPU v SoC Zynq, kde si pakety převezme ovladač v OS Petalinux.

Téma jsem si vybral z důvodu osobního zájmu ve všech částech práce – FPGA, počítačové sítě i OS Linux.

Práce je rozdělena na 3 kapitoly: Analýza, Návrh a implementace, Testování a verifikace. V první z nich popisuji teoretické koncepty potřebné k pochopení práce. V další části vysvětluji můj postup při tvorbě řešení. Poslední kapitola obsahuje způsob a výsledky testování práce.

Kapitola 1

Cíl práce

Hlavní cíl této práce je implementace IP jádra pro zpracování signálu z SFP klece, které podporuje rychlost 1G a 10G. Mezi nimi lze staticky nebo dynamicky přepínat. Vytvořené IP jádro také zahrnuje filtr, jenž odděluje provoz podle cílového portu. Vybraný provoz poskytuje na AXI Stream rozhraní pro jinou aplikaci v FPGA logice. Zbylé pakety posílá do paměti ARM procesoru, kde si je převezme správně nastavený ovladač v OS Petalinux.

V analytické části je cílem popsat použité technologie a odvodit z nich základní postup implementace a potřebné nastavení. Také vysvětluji později zmíněné a relevantní koncepty.

Předmětem praktické části je implementace. Nejdříve popisuji použitou platformu a její přípravu pro následující využití. Dále vysvětluji, jakým způsobem jsem implementoval zpracování SFP signálu, nejdříve v oddělených designech, potom s přepínáním v jednom IP jádře. Následuje návrh nově vytvořeného filtru portů a jeho spojení se zpracováním SFP do jednoho projektu.

V poslední kapitole je cílem simulovat nové moduly a otestovat jednotlivé části i celé IP jádro a popsat jejich výsledky.

Kapitola 2

Analýza

V této kapitole popisuji koncepty a protokoly použité v počítačových sítích a jaké platformy jsou vhodná pro zpracování síťového provozu. Také objasňuji, co znamená termín FPGA a jak takové zařízení funguje. Na závěr popisuji operační systémy a ovladače související s touto prací.

2.1 Počítačové sítě

Počítačové sítě zajišťují spojení mezi velkým množstvím digitálních zařízení. V této části popisuji protokoly, které určují, jak se zařízení v sítích identifikují a jakým způsobem komunikují.

2.1.1 Model ISO/OSI

Tento referenční model vznikl s cílem vytvořit standardy pro komunikaci mezi zařízeními nezávisle na výrobci či zařízení. ISO/OSI[1] vytvořilo IEEE (Institute of Electrical and Electronics Engineers, neboli Institut pro elektrotechnické a elektronické inženýrství). Obsahuje celkem 7 vrstev, pro tuto práci jsou důležité 4 z nich: fyzická, linková, síťová a transportní.

2.1.2 Fyzická vrstva

Zajišťuje fyzické prostředky pro navázání, udržení a ukončení fyzického spojení mezi dvěma entitami.

Cituji [2]: „Fyzické vrstvy se proto týkají standardy, které definují elektrické, mechanické, funkční a procedurální vlastnosti rozhraní pro připojení

různých přenosových prostředků a zařízení (tj. kabelů, modemů apod.) - tedy elektrické parametry přenášených signálů, jejich význam a časový průběh, vzájemné návaznosti řídících a stavových signálů, zapojení konektorů, a mnoho dalších parametrů technického i procedurálního charakteru. Úkolem entit fyzické vrstvy je pak na základě těchto standardů obsluhovat přenosové prostředky, připojené k příslušným rozhraním, a jejich prostřednictvím zajišťovat přenosy jednotlivých bitů."

2.1.3 Linková vrstva

Poskytuje spojení mezi síťovými entitami, včetně jeho navázání, udržení a ukončení. Jedno linkové spojení se může skládat z několika fyzických.

Linková vrstva[3] pak využívá prostředků fyzické vrstvy pro přenos větších bloků dat, označovaných jako rámce (frames). Pro adresování jsou definovány Media Access Control (MAC) adresy. U MAC adresy se předpokládá unikátnost, ta je zajištěna přidělením různé první části MAC adresy různým výrobcům[4], zatímco zbývající bity by měly výrobci nastavit jinak u každého zařízení, které vyrobí.

Na úrovni 2. vrstvy ISO/OSI běžně pracují domácí počítačové sítě, kde všechny zařízení mají přidělené IP adresy z jedné podsítě a pro komunikaci by teoreticky stačila pouze MAC adresa.

2.1.4 Síťová vrstva

Cituji [5]:

„Chtějí-li spolu komunikovat dva uzly počítačové sítě, mezi kterými neexistuje přímé spojení, je nutné pro ně najít alespoň spojení nepřímé - tedy vhodnou cestu, vedoucí přes mezilehlé uzly od jednoho koncového uzly ke druhému. Možných cest může být samozřejmě více, někdo je však musí najít, jednu z nich vybrat, a pak také zajistit správné předávání dat po této cestě. Všechny tyto úkoly má v referenčním modelu ISO/OSI na starosti síťová vrstva."

Na úrovni 3. vrstvy pracují routery (směrovače), které zajišťují zmíněné úkoly na základě IP adres a data sdružují do tzv. paketů. Přesunují data napříč sítěmi 2. úrovně, což umožňuje komunikaci takřka s kýmkoliv kdekoliv díky celosvětové počítačové síti zvané Internet.

2.1.5 Transportní vrstva

Transportní vrstva řeší problém nespolehlivé síťové vrstvy – výpadky dat, zpomalení, apod. – nebo alespoň může řešit.

Nejlepším příkladem protokolu pracující na transportní vrstvě je TCP (Transmission control protocol[6]). Ten vytváří spojení mezi koncovými stanicemi, řadí pakety podle odeslaného pořadí, kontroluje doručení paketů a stará se o jejich opětovné poslání. Hlavičky TCP obsahuje mimo jiné hodnotu zdrojového a cílového portu. Používá se pro většinu aplikací, které se ke spojení mohou chovat jako k extrémně spolehlivému za cenu mírně větší režie.

Naproti tomu UDP (User datagram protocol[7]) žádné záruky neposkytuje, k informacím v paketu přidává pouze nejnutnější údaje tj. čísla portů (stejně jako TCP), délka a kontrolní součet. UDP má specifické zaměření – jeho hlavička má minimální velikost, neplýtvá se zdroji pro odeslání potvrzení a neztrácí se čas řazením nebo opětovným posíláním. To je vhodné pro aplikace, které potřebují maximální rychlost nebo minimální odezvu, např. stahování/odesílání dat, nebo video hovory. Pokud se musí doručit všechna data, často je výhodnější, aby to bylo řízeno samotnou aplikací.

2.2 Ethernet

Ethernet je soubor standardů vyvinutý IEEE pod označením 802.3[8]. Mimo jiné popisuje různé rychlosti a média, po kterých se posílá signál.

Základní vlastností Ethernetu je přenos po elektrických vodičích nebo optických vláknech, na rozdíl od Wi-Fi, kde se data přenáší vzduchem pomocí elektromagnetických vln. Ethernet je standard jak první vrstvy ISO/OSI, tak částečně druhé.

Části síťového spojení podle standardu 10GE WAN PHY[9]:

PMD

Zde probíhá komunikace se zařízením přenášejicí analogové signály pomocí MDI (Media independent interface) a s PMA. Pro 10 gigabit jsou oba směry uspořádány po 2 oktetech (celkem 16 bitů) a pracují na frekvenci 622.08 Mhz.

PMA

Physical Medium Attachment provádí synchronizaci PMA rámců a za-

balení oktetů, scrambling/descrambling PMA rámců pomocí $x^7 + x^6 + 1$ synchronního scrambleru a přenos výsledných dat z a do PMD podčásti fyzické vrstvy.

PCS

Physical Coding Sublayer, jak již název napovídá, zajišťuje kódování/dekódování, scrambling/descrambling a další drobné operace.

MAC

MAC zabaluje data z vyšších vrstev ISO/OSI modelu do rámců (frames) a přidává všechny potřebné údaje, také provádí opačné operace při příjmu dat. Některé z těchto operací jsou rozpoznání rámců, vytváření kontrolních součtů a zahození chybných rámců.

LLC

Logical link control je vyšší část 2. (linkové) vrstvy. Pro 802.3 je LLC volitelná součást. Vytváří rozhraní mezi MAC a síťovou vrstvou pomocí multiplexování umožňující přenos různých protokolů na stejném rozhraní. Může obsahovat obsluhu pro flow control a automatické posílání žádostí o přeposlání ztracených nebo zahozených rámců.

2.2.1 Vnější PHY

Fyzická vrstva nemusí být obsažena na stejném integrovaném obvodu jako další vrstvy ISO/OSI. Ke komunikaci mezi první a druhou vrstvou slouží MII (Media independent interface), který podle názvu není závislý na konkrétním fyzickém médiu. To se používá na transcieverech, které mohou využívat různé technologie přenosu (optické nebo elektrické spojení).

2.2.2 SFP

SFP[10] rozhraní nejčastěji připojuje optická vlákna, ale může se použít i k zapojení měděných vodičů. Toto rozhraní je vysoce modulární, umožňující osazení libovolným transceiverem podporující i vyšší rychlosti, díky čemuž lze snadno vylepšit či změnit existující síťovou infrastrukturu.

SFP rozhraní poskytuje krom datových signálů dvouvodičové datové a hodinové signály pro komunikaci s moduly a také několik kontrolních:

`Tx_Fault` – indikace poruchy transceiveru

`Tx_Disable` – deaktivace optického výstupu

`MOD_Abs` – detekce chybějícího modulu

`Rx_LOS` – oznámení ztráty signálu receiveru

SFP moduly, které se zapojují do tohoto rozhraní, mohou, ale nemusí, obsahovat vlastní transceiver. Nutný je například v případě komunikace přes optická vlákna, kde se signál překládá z elektrických signálů na světelné.

Cituji [11]: „Všechny SFP moduly obsahují paměť EEPROM, dostupnou na standardizované I^2C adrese ve standardním formátu, což dovoluje hostitelskému systému zjistit, jaké SFP moduly jsou připojeny a jaké jsou jejich schopnosti."

Vzhledem k faktu, že SFP bylo plánováno jako rozhraní pro servery a infrastrukturu, SFP poskytuje možnost měnit moduly za běhu systému, protože není přijatelné restartovat takové systémy kvůli pouhé změně rozhraní.

2.3 Platformy pro síťové programy

Aplikace pro zpracování síťového provozu lze spustit na různých typech zařízení, od běžných po vysoce specializované, výhody a nevýhody některých možností vysvětluji níže.

2.3.1 Tradiční počítače

Nejjednodušší možností pro většinu osob, které chtějí pracovat se síťovým provozem, bude klasický počítač. Za prvé z důvodu dostupnosti (nějaký počítač pravděpodobně vlastní), za druhé z důvodu jednoduchosti zprovoznění a nastavení aplikace. Tradičním počítačem zde označuji systémy s procesorem architektury x86 nebo ARM, to jest jak stolní PC, notebooky, tak i servery nebo mobilní zařízení.

Programy pro tyto zařízení se pravděpodobně dají najít v open source verzi na internetu, a pokud přesně nesplňují požadavky, lze programy snadno upravit v nějakém z populárních programovacích jazyků. Ani vytvoření úplně nového programu pro specifické účely není příliš náročný úkol s knihovnami vyvinutými přímo k tomuto účelu, například Scapy pro Python [12].

Další výhodou počítačů s rozsáhlým operačním systémem je existující implementace základních síťových protokolů například pro získání IP adresy (DHCP) nebo podporu směrování (ARP).

Počítače jsou univerzální, což jim dává skvělou flexibilitu, ale také to znamená poměrně nízkou efektivitu. Pokud výkon všestranného počítače nestačí nebo nesplňuje požadavky na cenu či spotřebu, je nutné se obrátit k jiným platformám.

2.3.2 Vestavná zařízení

Vysokou efektivitu spolu s často ještě nižší složitostí zajišťují vestavná zařízení.

Jsou vyrobena velmi specificky, díky tomu nabízejí řádově vyšší efektivitu v porovnání s PC. Dokáží poskytovat několik různých služeb v rámci daného účelu, například router a switch zároveň, ale změna účelu zařízení není podporována výrobcem, pokud není přímo znemožněna.

V oblasti telekomunikací se nejčastěji jedná o switch nebo router. Switch propojuje zařízení uvnitř jedné sítě, často má velké množství portů pro připojení klientů. Router zajišťuje rychlé spojení napříč sítěmi, například mezi lokální sítí a Internetem.

2.3.3 FPGA

Alternativou pro zákazníky hledající vysokou efektivitu jsou také FPGA zařízení. FPGA poskytují bezkonkurenční flexibilitu díky jejich unikátní struktuře, díky které se dokážou přizpůsobit jakémukoliv účelu.

V případě využití v oblasti telekomunikací poskytují velmi vysoký výkon, jsou omezeny spíše existujícími protokoly a rychlostí rozhraní na desce – existují FPGA s podporou rychlosti až 400 gigabit/s[13], v současné době nejvyšší definovaná rychlost standardu Ethernet[14]. Dávají uživateli úplnou kontrolu nad každým stádiem zpracování dat ze sítě, tím umožňují vytvoření kompletně proprietárních aplikací a protokolů nebo podporu pouze potřebných částí, bez plýtvání paměti či zdrojů.

Tato vysoká flexibilita je také nevýhodou, znamená to nutnost manuálně zajistit obsluhu všech nutných částí síťových protokolů. Existují IP jádra pro zpracování základních vrstev ISO/OSI (např. [15] [16]), ale ta bývají licencovaná a uživatel stále musí implementovat zpracování vyšších vrstev.

2.4 Technologie FPGA

Field-programmable gate array (FPGA) je programovatelný integrovaný obvod, který lze přenastavit podle potřeby. I přes to jsou řádově efektivnější než běžné počítače díky jejich unikátní architektuře.

Převzato z [17], překlad od [18]: „Srdcem každého FPGA je programovatelná struktura, která je reprezentována jako řada programovatelných logických bloků. Každý z těchto logických bloků obsahuje vyhledávací tabulku (LUT), multiplexor a registr. Vše lze nakonfigurovat (naprogramovat) tak, aby fungovaly dle potřeby.“

Takové možnosti znamenají snadné a velice flexibilní přizpůsobení se měnícím se podmínkám, například nové verze zabezpečení nebo stav prostředí, proto jsou FPGA použity mimo jiné ve vesmírných sondách [19].

2.4.1 Proces vývoje pro FPGA

FPGA se běžně navrhují v jazyce HDL – hardware description language, neboli jazyk popisu hardware, dva nejpoužívanější se nazývají VHDL a Verilog. V nějakým aspektech jsou podobné programovacím jazykům, také se zde objevují proměnné a direktivy jako `if` a `else`, ale kvůli odlišnému způsobu, jakým pracují FPGA oproti procesorům (např. architektury ARM), se v základních aspektech liší.

Fundamentálním stavebním blokem designu je proces (název z VHDL). Ten může být složitý (implementace sčítání, násobení a indexování zároveň) nebo triviální (přiřazení hodnoty). Největším rozdílem oproti programům pro běžné procesory je fakt, že všechny procesy se spouští a běží zároveň, pokud není určeno jinak.

Když se HDL používá pro ASIC, design přesně popisuje reálná logická hradla a registry. Zatímco FPGA se pouze chovají jako daný ASIC díky vyhledávacím tabulkám (LUT, lookup table). K nastavení LUT tak, aby fungovaly podle naplánovaného designu, se používá tzv. bitstream.

Pro jeho vytvoření se musí provést 3 základní fáze vývojového procesu (podle [20], výňatek z [21]):.

- syntéza,
- rozmístění a zapojení,

- vytvoření bitstreamu.

Během syntézy se projekt popsaný HDL jazykem přemění do logické formy pro vhodné zpracování v další fázi. Tento výstup se nazývá netlist (lze přeložit jako seznam sítí). Netlist popisuje, které elementy design obsahuje, jak jsou mezi sebou propojeny, jejich navázání na vstupní a výstupní piny FPGA, apod. Neurčuje jejich fyzickou pozici v obvodu, k tomu slouží další fáze.

Fáze rozmístění a zapojení, jak již název napovídá, nejdříve plánuje, kde budou logické členy z netlistu umístěny a následně jak budou reálně zapojeny. Během tohoto procesu se musí dbát na omezení (constraints), které mimo jiné diktují k jakým pinům na desce se musí design připojit a především frekvenci, na které mají části designu pracovat.

Frekvence přímo určuje periodu, tedy čas mezi hodinovými cykly, během které musí všechny kombinační obvody dokončit změnu své polohy. To je kritické z důvodu ustálení signálu na vstupu registru, který signál ukládá do dalšího hodinového cyklu. Pokud by se tak nestalo a vstup by nebyl stabilní, výstup z registru by mohl nabýt jakékoliv hodnoty, což by způsobilo řetězovou reakci a celý design by mohl selhat.

Splní-li se omezení, pokračuje se další, poslední fází: generování bitstreamu, doslova přeloženo jako proud bitů. Bitstream je soubor, který se nahraje do FPGA a tím nastaví logické obvody uvnitř do podoby, která přesně simuluje výsledky z předchozích částí vývojového procesu.

2.4.2 Protokol AXI

K datové komunikaci mezi různými částmi Xilinx FPGA se často používá standard Advanced eXtensible Interface[22] (AXI). Jeho hlavní výhody zahrnují rychlost komunikace, volitelnou šířku sběrnice, široké možnosti přizpůsobení a případně také snadnou implementaci v závislosti na podporovaných funkcích. Komunikace probíhá synchronně s přivedeným hodinovým taktem a s každou náběžnou hranou lze přenést data. V použitých IP jádrech pro zpracování paketů a komunikaci mezi procesorem a FPGA logikou neexistuje jiná možnost než komunikovat přes AXI rozhraní, proto i moje výsledné IP jádro bude komunikovat stejným způsobem.

Přenos dat probíhá převážně jedním pevně nastavených směrem (od master ke slave). Základní signály pro komunikaci (krom datových) jsou pojme-

novány TVALID a TREADY. Pomocí TVALID master oznamuje slave, že má připravena data k odeslání a vystavuje je na datové připojení. TREADY naopak ovládá slave, tímto signálem dává najevo, že je připraven přijímat data.

2.4.2.1 AXI4-Stream

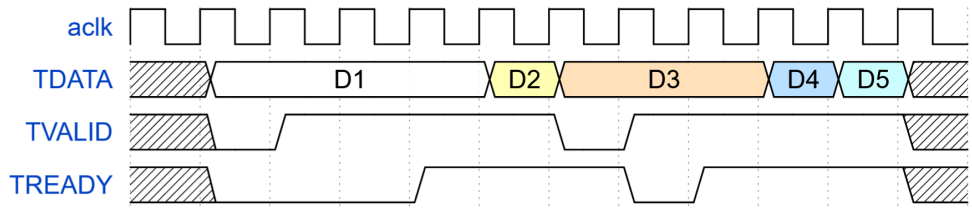
Minimální verze AXI4 se nazývá AXI4-Stream. Vyžaduje pouze datový kanál a signál TVALID. V takovém případě master předpokládá, že slave dokáže vždy přijmout data, jako by TREADY bylo nastaveno permanentně na 1.

Volitelné část zahrnuje TDEST a TID pro určení cíle, TLAST označující poslední AXI transakci příslušící současnému proudu, což v této práci nejčastěji znamená paket. Také lze využít TKEEP a TSTRB pro označení platných bytů uvnitř přenesených dat.

Hlavní datový kanál je pouze jeden, tato verze je určena pro jednoduché aplikace bez složité adresace.

Komunikace probíhá poměrně jednoduše, po nastavení obou kontrolních signálů (TVALID a TREADY) na logickou 1 se data považují za přenesená, další hodinový cyklus ztrácejí platnost a mohou se přenést jiná data.

Kombinace načasování signálů může vyústit ve 3 různých situacích (ukázka na diagramu 2.1, [23]) : TVALID bude nastaven dříve než TREADY. V tomto případě musí master držet TVALID na 1, dokud nezaznamená TREADY na 1. Pokud naopak slave nastaví TREADY na 1, dokud TVALID od mastera zůstává na 0, může libovolně měnit hodnotu TREADY. Nebo mohou obě strany nastavit svoje signály na 1 ve stejný hodinový takt, potom v daném cyklu proběhne přesun dat a pokud obě strany podrží kontrolní signály na 1, v následujícím se pošlou jiná data, tímto způsobem lze dosáhnout teoretické rychlosti rovné frekvence $\times$ šířka sběrnice.



Obrázek 2.1: Ukázka komunikace po AXI Stream rozhraní

2.4.2.2 AXI4

AXI4 se liší oproti AXI Stream především množstvím signálů a možností. Skládá se z několika téměř nezávislých kanálů pro data, každý z nich funguje stejným způsobem jako AXI Stream.

Definuje kanály Address Read (AR), Read (R), Address Write (AW), Write (W) a Write Response (B). Dvojice AW a W slouží k zápisu dat – nejdříve se na AW přenese adresa zápisu, potom se pomocí W kanálu data zapíší. Obdobně se pracuje s AR a R, ale jedná se o čtení dat z určené adresy. Kanál B slouží pro komunikaci odpovědi na zápis. Čtení dat a potrvzení zápisu jsou jedny z vyjímek, kdy slave posílá data směrem k master.

Hlavní změnou oproti AXI Stream je možnost přímé adresace (na AR a AW). Také podporuje burst mód. Burst (česky dávka) znamená, že na jednu žádost po zapsání adresy slave odpoví několika přenosy na hlavním datovém kanále.

Od AXI je odvozena AXI4 Lite. Rozdíl je minimální, AXI4-Lite pouze přidává několik restrikcí za účelem snížení složitosti implementace. Nejdůležitější omezení jsou: každý burst obsahuje pouze 1 transakci a všechny datové transakce probíhají na celé šířce datové sběrnice (pouze 32 nebo 64 bitů). Z hlediska adresace a přesunů dat se chová stejně jako AXI4.

2.4.3 FPGA - GTH/GTX Transceivery

GTX jsou multi-gigabit transceivery, jenž jsou velmi kvalitně popsány v knize High-Speed Serial I/O Made Simple[24]:

„Vstup, neboli referenční hodiny, pro Multi-Gigabit Transceiver (MGT) má velmi přísné požadavky. To zahrnuje přísný požadavek na frekvenci, běžně specifikován v povolených částech na milion (parts per million, PPM) chyby

frekvence. Také bude mít přísné požadavky na jitter definované v podobě jednotek času (pikosekundy) nebo jednotkových intervalů (UI).

Tak přísné požadavky umožňují funkci PLL a obvodů pro extrakci hodin. Toto často vyžaduje přesný krystalový oscilátor na každém plošném spoji v systému, který využívá MGT. Tyto krystalové oscilátory jsou o úroveň výše oproti většině používaných pro digitální systémy a budou dražší. V mnoha případech, čipy generující hodiny a PLL mají příliš mnoho jitteru k použití."

Pro časování SFP transceiverů použitých v této práci není možné využít běžné oscilátory v FPGA nebo PLL na SoC Zynq, kvůli nedostatečné kvalitě generovaného signálu. Je nutné použít oscilátory a transceivery speciálně určené k přímo tomuto účelu, které Xilinx v této řadě FPGA nazývá GTX. Tyto transceivery na desce ZC706 podporují rychlosti od 500 Mb/s do 12,5 Gb/s ([25], str. 21).

2.4.4 Zynq - Kombinace technologií ARM a FPGA

Systémy na čipu (SoC) Zynq[26] obsahují jak ARM procesor, tak FPGA část. Toto řešení poskytuje výhody obou technologií, zákazník může využít obrovské softwarové podpory dostupné pro OS Linux, který lze na čipu spustit, nebo psát vlastní programy v běžných programovacích jazycích (C, C++, Python, atd.), a zároveň zpracovávat obrovské množství dat velmi rychle a efektivně v FPGA části.

Tento systém je ideální pro aplikace pracující se síťovým provozem, protože základní funkce potřebné pro síťové připojení jsou často již implementovány v OS, či je lze snadno přidat, zatímco náročné výpočty citlivé na zpoždění (např. audio či videohovory) lze přenechat FPGA.

Ke komunikaci mezi programovatelnou logikou a CPU slouží několik AXI rozhraní. Pro přenosy větších objemů dat do paměti procesoru se používá rozhraní HP (high performance), ty jsou na desce umístěny celkem 4. GP (general purpose) AXI rozhraní jsou využívány pro odesílání dat z procesoru k FPGA části, například nastavení nebo vypnutí/zapnutí částí programovatelné logiky.

2.4.4.1 Platforma ZC706

ZC706 Evaluation board poskytuje potřebné části k vytvoření této práce:

- Zynq-7000 SoC

- GTX transceiver
- konektor Small form-factor pluggable plus (SFP+)
- Ethernet PHY RGMII rozhraní s konektorem RJ45
- rozhraní USB JTAG přes Digilent modul s mikro-B USB konektorem
- I²C
- programovatelný LVDS oscilátor (diferenciální)

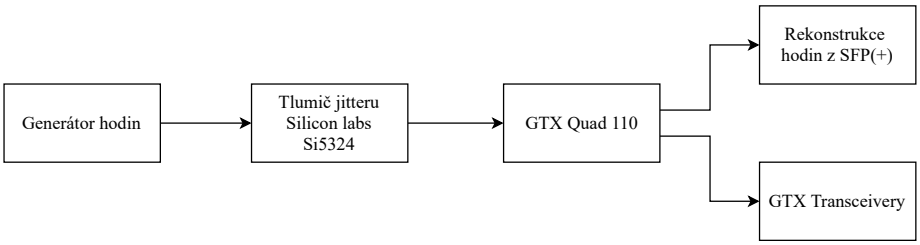
Správná funkce SFP pro rychlost 10G i 1G vyžaduje vlastní hodinový signál.

2.4.5 Generování hodin

Na desce ZC706 poskytuje několik možností generování hodin pro běžnou programovatelnou logiku, např. PLL obvody na SoC, ale transceivery pro SFP mají velmi přísné požadavky na kvalitu hodinového signálu, což zabraňuje použití běžných oscilátorů. V této práci bylo možné zvolit pouze generátor Silabs Si5324[27], protože právě tento dokáže ztlumit jitter dostatečně, aby byl vhodný pro GTX transceiver SFP konektoru. Další výhodou je možnost změnit frekvenci za běhu systému, což je nutné pro dynamickou změnu rychlosti. 10G vyžaduje frekvenci 156,25 Mhz, zatímco 1G rozhraní pracuje na 125 Mhz. Obě frekvence jsou v rozsahu frekvencí podporovaných oscilátorem.

2.4.6 FPGA připojení SFP 1G/10G na ZC706

Deska ZC706 [28] obsahuje tlumič jitteru U60 Silicon Labs Si5324 na zadní straně desky. V SoC logice lze implementovat obvod pro generování hodin a tento signál přivést na diferenciální pár pinů (AD20 a AE20) pro tlumení jitteru pomocí Silabs Si5324. Výstup z tohoto čipu je veden jako referenční hodiny do vstupu GTX Quad 110 (AC8, AC7). Hlavní účel tohoto signálu je podpora aplikací, které provádí rekonstrukci hodin z SFP(+) modulu, a použití jako referenční hodiny GTX transceiveru. Znázorněno na 2.2.



Obrázek 2.2: Průběh signálu pro GTX Quad 110

2.5 Software výbava pro síťové aplikace

2.5.1 Vlastní aplikace

Vždy existuje možnost vytvořit si vlastní ekvivalent operačního systému pro konkretní zařízení. Tento postup však vyžaduje velké množství času a zdrojů, ale i tak pravděpodobně nebude stejně spolehlivý jako ostatní možnosti prověřené množstvím uživatelů a řadou let.

2.5.2 RTOS

Operační systém reálného času (Real Time Operating System, RTOS) zajišťuje obsluhu systému a garantuje včasnou odpověď na změnu signálu. Obecně systém reaguje velmi rychle, což vede k minimálnímu zpoždění, které je zásadní pro real-time přenos, například videohovor.

Implementace síťových protokolů může být vlastní, pokud je daná práce něčím specifická. Toto vyžaduje velké množství práce a testování, proto pro obsluhu obecného síťového provozu bude nejspíš vhodnější využít existující open-source řešení.

Možnosti zahrnují lwIP [29], pokud stačí samotná implementace TCP/IP protokolů a již existuje vlastní RTOS, nebo lze zvolit operační systém se zabudovanou podporou síťového provozu, například FreeRTOS, Zephyr, či MynewtApache.

2.5.3 Embedded Linux

Alternativním řešením je OS Linux přizpůsobený pro dané zařízení, ale také má nejvyšší nároky na velikosti pamětí a rychlost procesoru. I přes to, že některé funkce plného operačního systému nepodporuje, obsahuje všechny základní funkce s možností snadného doinstalování rozšíření nebo napsání vlastních programů a ovladačů, ale především pro tento účel má implementovány všechny nutné části síťového komunikace

2.6 Linux

Linux byl vyvinutý jako OS pro PC z původně serverového OS UNIX. Jednou z jeho základních vlastností je, že kód jádra systému je veřejně dostupný (open source) a kdokoliv ho může zkontrolovat a zefektivnit (pokud to schválí určení vývojáři), díky tomu je Linux velmi optimalizovaný. Také je možné si vytvořit novou část, nebo změnit či odebrat existující – pro svoje použití bez schválení. Tyto vlastnosti vedly k rozšíření operačního systému Linux na obrovské množství zařízení různých typů, včetně těch vestavných – např. Xilinx vytvořil distribuci Linuxu s názvem Petalinux pro svoje zařízení Zynq (mimo jiné).

2.6.1 Ovladače

Ovladače v Linuxu existují ve dvou formách[30]: monolitické (kompilované spolu s kernelem) a modul kernelu. Ovladač jako modul má výhodu, že může být přidán nebo odebrán za běhu systému, díky čemuž v případě chyby může kernel pouze odebrat modul, který vyvolal danou chybu, místo toho, aby celý systém selhal.

Lze je také rozdělit na různé druhy podle jejich chování[31]. K většině jde přistupovat jako k souborům – jde z nich číst nebo do nich zapisovat, chovají se jako proudy (streams). Někdy je možné pracovat s jednotlivými byty, jindy s bloky (např. disk a jeho sektory), ale princip chování zůstává stejný. Zařízení s těmito ovladači se namapují jako soubory do souborového systému.

Oproti tomu síťová zařízení pracují pouze s pakety. Není možné je jednoduše mapovat jako soubory, proto nemají cestu v souborovém systému, pouze unikátní jméno (např. eth1). Jádro systému s takovými ovladači komunikuji pomocí funkcí, které dávají smysl pro síťová zařízení.

2.6.2 Device tree

Open hardware device tree, nebo jednoduše Device tree, je datová struktura a jazyk pro popis hardware srozumitelný pro OS. Operační systém si tuto strukturu přeloží a podle ní provede vhodné nastavení a načte správné ovladače.

Jak název napovídá, device tree jako datová struktura má formu stromu[32]. Díky tomu lze snadno přidávat nové node (uzly), které mohou být definovány jako samostatné, nebo je zahrnout pod již existující jako potomky, pokud k nim mají patřičný vztah, například přidání nového zařízení na již existující sběrnici.

Device tree není součástí kernelu, načítá se v oddělených souborech, každý z nich může obsahovat různé části device tree, např. oddělený soubor pro uživatelem přidaná zařízení jako v Petalinux.

2.6.3 Petalinux

Petalinux je distribuce OS Linux, která byla vyvinutá a testovaná pro zařízení Xilinx. K nastavení a kompilaci tohoto systému se používá Petalinux Tools. S těmito nástroji je možné přidávat/odebírat různé součásti systému, jako ovladače a programy, nebo vytvářet vlastní. Také zajišťují kompilaci celého systému včetně uživatelských aplikací, tak aby šel OS bez problému spustit na cílovém zařízení. Petalinux tools pro sestavení systému k bootování používá Yocto Project.

Petalinux tools nabízí také možnost přidání libovolných node do device tree vytvořeného systému. To je nutné pro umožnění funkce některých IP jader od Xilinx, např. DMA nebo zpracování signálu ze sítě.

K nastavení komponent obsažených ve vytvořeném OS nabízí Petalinux tools grafické uživatelské rozhraní. To spouští na 3 úrovních: základní pro aktivace existujících možností, rootfs (souborový systém) k přidání vlastních knihoven a aplikací nebo kernel (jádro) pro nastavení a zahrnutí ovladačů v samotném jádře systému.

Kapitola 3

Návrh a Implementace

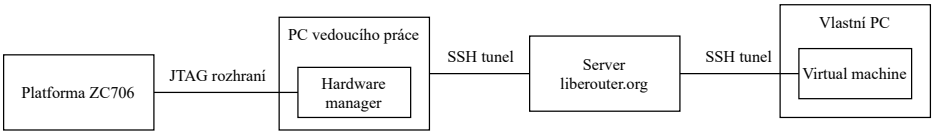
V této kapitole vysvětluji, jak jsem implementoval IP jádro pro zpracování a filtrování síťového provozu. Nejdříve popisuji platformu, pro kterou jsem práci vytvářel. Další část se týká konfigurace OS Petalinux a vytvoření designu pro zpracování SFP signálu v programu Xilinx Vivado s využitím existujích IP jader. Následující kapitola opahuje detailní popis mého vlastního modulu pro filtrování AXI Stream provozu. V závěru kapitoly vysvětluji, jakým způsobem jsem předchozí design spojil s mým modulem.

3.1 Platforma ZC706

Podpora v software nebyla problém – pro ZC706 existuje přímo od výrobce BSP (board support package), který nastaví Petalinux tools tak, aby byly kompatibilní s danou platformou. Xilinx Vivado nezahrnuje tuto desku v základní verzi (Webpack edition), pouze ve vyšší verzi (System edition). Ta je nainstalována na CESNET serveru, kam jsem dostal přístup, proto jsem většinu práce s designem prováděl vzdáleně. FPGA jsem programoval skrze JTAG rozhraní (velkou část práce přes internet pomocí SSH tunelu, znázorněno na diagramu 3.1).

3.1.1 Nastavení hodin

Nejdříve jsem chtěl nastavit frekvenci staticky pro ověření správnosti. K tomu jsem v device tree operačního systému Petalinux přidal několik node (zahrnuty



Obrázek 3.1: Diagram programování desky ZC706 na dálku

ve výpisu kódu 3.1) podle [33]. Konfiguraci jsem si později ověřil spuštěním designu pro zpracování 10G.

3.1.2 Komunikace s SFP modulem

Pro přepínání mezi rychlostmi 1 Gbit a 10 Gbit za běhu FPGA jsem potřeboval zprovoznit komunikaci s SFP modulem přes rozhraní I^2C, pomocí kterého mohu oscilátor nastavit. V OS Linux již k tomuto účelu byl vytvořen ovladač zahrnutý v Petalinux. CESNET má vytvořený program pro nastavení oscilátoru využívající tento ovladač – zápisem do souboru změní frekvenci.

Pro zprovoznění I^2C komunikace mezi OS a oscilátorem přidal node to device tree v souboru system-user.dtsi (3.2). V petalinux tools na úrovni rootfs musí být povoleno `i2c-tools`. V kernelu OS je vyžadován zapnutý Xilinx I2C controller (`I2C_XILINX`) [34]. Poté oscilátor komunikoval s OS podle výpisu zpráv z kernelu.

Dále jsem do mé instance Petalinux přidal dříve zmíněný program pro snadné nastavení oscilátoru od CESNET. Program fungoval podle očekávání a nastavil oscilátor na zadanou hodnotu.

3.2 Návrh IP jádra

Nejdříve jsem implementoval zpracování SFP signálu bez filtru, v této sekci popisuji, jakým způsobem jsem dosáhl funkčních designů. Začal jsem 10G verzí, protože její implementace se zdála méně komplikovaná.

3.2.1 10G

Pro 10G Ethernet na desce Xilinx ZC706 jsem našel vypracovaný projekt[35], který byl bohužel vytvořen v jiné verzi software, takže nebylo možné jednoduše

```
/include/ "system-conf.dtsi"
/ {
    refhdmi: refhdmi {
        compatible = "fixed-clock";
        #clock-cells = <0>;
        clock-frequency = <114285000>;
    };
    oscface-01 {
        compatible = "cesnet,oscface";
        clock-names = "si5324_a";
        clocks = <&si5324 0>;
    };
};

&i2c0 {
    i2c-mux@74 {
        i2c@4 {
            si5324: clock-generator@68 {
                status = "okay";
                compatible = "silabs,si5324";
                reg = <0x68>;
                #address-cells = <1>;
                #size-cells = <0>;
                #clock-cells = <1>;
                clocks = <&refhdmi>;
                clock-output-names = "si5324_a";
                clock-names = "xtal";

                jitter_clk: clk0 {
                    reg = <0>;
                    clock-frequency = <125000000>;
                };
            };
        };
    };
};
```

Výpis kódu 3.1: Node oscilátoru Si5324 v device tree

```
&i2c0 {
    i2c-mux@74 {
        [...]
        i2csfp: i2c@0 { // i2c SFP
        };
    };
};
```

Výpis kódu 3.2: I^2C node v device tree k programování SFP

spustit přiložené skripty, ale dokázal jsem podle nich vytvořit funkční block design v Xilinx Vivado, popsáno dále.

Do nově vytvořeného block designu jsem přidal IP jádra Zynq7 Processing System (PS), reprezentující procesor na SoC Zynq a jeho rozhraní, a 10G Ethernet Subsystem zajišťující zpracování paketů z fyzické úrovně.

Pro komunikaci mezi těmito moduly slouží AXI Direct Memory Access (DMA) umožňující pakety ze subsystému ukládat přímo do paměti RAM na SoC, kde s těmito daty dále může pracovat ovladač v OS. Propojení mezi zmíněnými IP jádry zajišťují 2 instance AXI Interconnect, každá spojující několik AXI rozhraní, která jsou využívány k datovým přenosům – první od CPU k DMA a subsystému, druhá z DMA do paměti.

V subsystému jsem změnil nastavení frekvence na 156,25 MHz odpovídající specifikaci a zvolil sdílenou logiku v jádře (Include shared logic in core). V Zynq PS jsem přidal rozhraní AXI HP0 pro vysokorychlostní přenos paketů do paměti a hodiny FCLK0, které jsou připojeny k části jednoho AXI Interconnect, jednomu AXI rozhraní na DMA a oběma AXI na Zynq7 PS.

Dále jsem do designu přidal tato Xilinx IP jádra:

- Processor System Reset pro korektní nastavení signálu reset pro AXI rozhraní ovládané Zynq PS,
- 2 AXI4-Stream fronty (AXI4-Stream Data FIFO), které podle zmíněného projektu zabraňují deadlocku,
- Utility Vector Logic nastavený na negaci pro zajištění správné hodnoty resetu,

```
&gem1 {
    local-mac-address = [00 0a 35 00 00 01];
    phy-mode = "rgmii-id";
    status = "okay";
    xlnx,ptp-enet-clock = <0x69f6bcb>;
    phy-handle = <&phy1>;

    phy1: phy@1 {
        compatible = "Xilinx PCS/PMA PHY";
        device_type = "ethernet-phy";
            xlnx,phy-type = <5>;
            reg = <1>;
    };
};
```

Výpis kódu 3.3: 10G transceiver node v device tree

- Concat pro spojení přerušení do Zynq PS do jednoho signálu a
- 2 konstanty (Constant) pro správné nastavení 10G subsystému.

Toto zapojení jsem otestoval samostatně, než jsem k němu přidal modul filtru. Výsledné zapojení s filtrem je znázorněno na obrázku 3.2, obsahuje pouze důležité datové cesty.

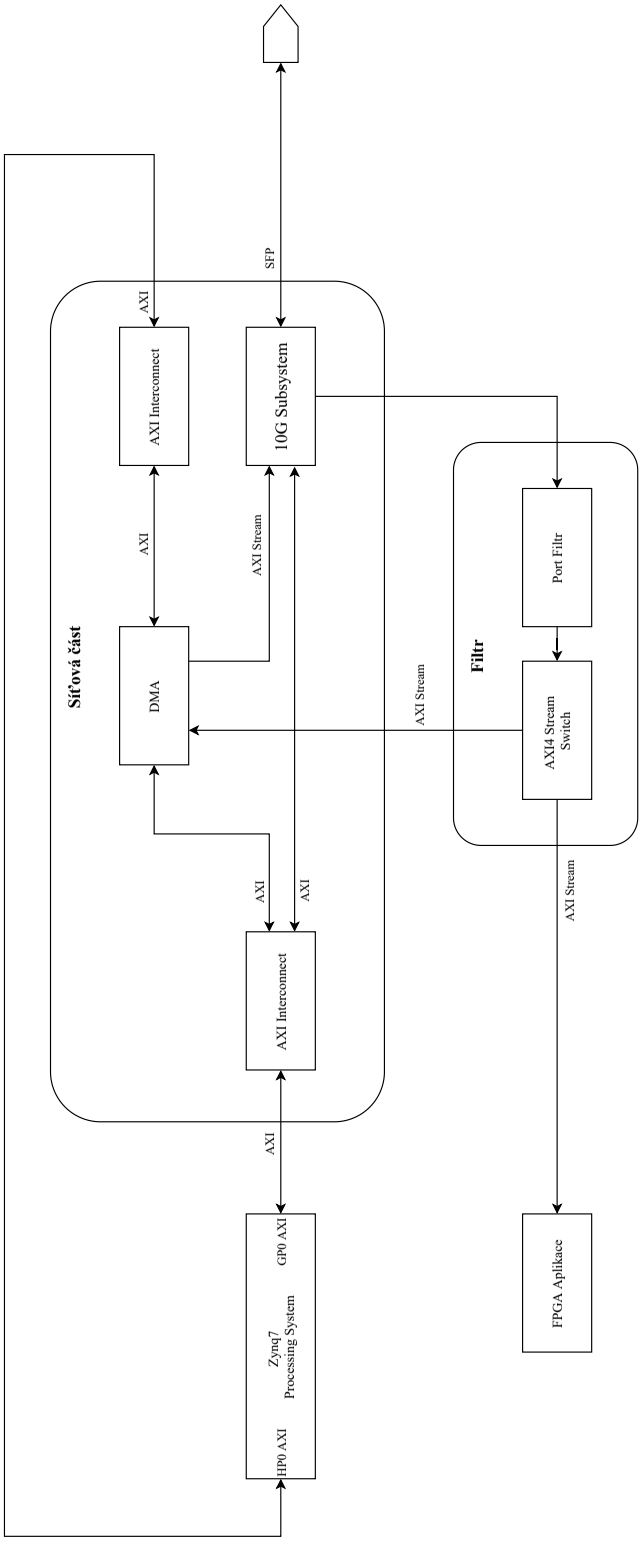
Rozsah adres v Address editoru jsem nastavil podle skriptu z projektu, offset se nastavil automaticky. Tento block design jsem ověřil pomocí Verify Design, potom jsem spustil jsem syntézu, implementaci a nechal vytvořit bitstream.

Následně jsem přidal node (ve výpisu kódu 3.3) do device tree podle [36]

Do Petalinux jsem zahrnul potřebný ovladač (`Drivers for xilinx PHYs`) do jádra Petalinux a nakonfiguroval OS pro desku ZC706.

Po sestavení nové verze Petalinux a jejím nahrání na desku bylo síťové rozhraní rozpoznáno a při zadání příkazu `ifconfig -a` se zobrazilo jako `eth1`. Poté jsem nastavil IP adresu a masku a aktivoval rozhraní pomocí `ifconfig eth1 192.168.88.123 netmask 255.255.255.0 up`.

Rozhraní jsem dále testoval, postup a výsledky popsány v kapitole Testování.



Obrázek 3.2: Zapojení modulů pro rychlost 10G

Uvádím pouze AXI a SFP rozhraní, vícenásobná spojení jsou reprezentována jednou šipkou.

3.2.2 1G

Zprovoznění 1G Ethernet na desce ZC706 bylo značně komplikovanější. Potřeboval jsem využít 3 základní IP jádra od Xilinx: Tri Mode Ethernet MAC (TEMAC), AXI Ethernet buffer a 1G/2.5G Ethernet PCS/PMA or SGMII (PCS/PMA). První 2 v těchto IP jáder jsou běžně zabalena uvnitř jednoho IP jádra AXI 1G/2.5G Ethernet Subsystem, ale tento subsystém neposkytuje přístup k rozhraní AXI Stream mezi Ethernet buffer a TEMAC. Právě na toto rozhraní potřebuji umístit filtr, proto jsem nemohl subsystém použít a musel jsem ho sestavit z jednotlivých součástí. Ethernet buffer nelze přidat přímo do designu, musel jsem ho zkopírovat z AXI 1G/2.5G Ethernet Subsystem. Podle subsystému jsem zapojil bloky Ethernet buffer a TEMAC, za ně připojil PCS/PMA, ke kterému jsem připojil několik konstant pro nastavení několika vstupních hodnot.

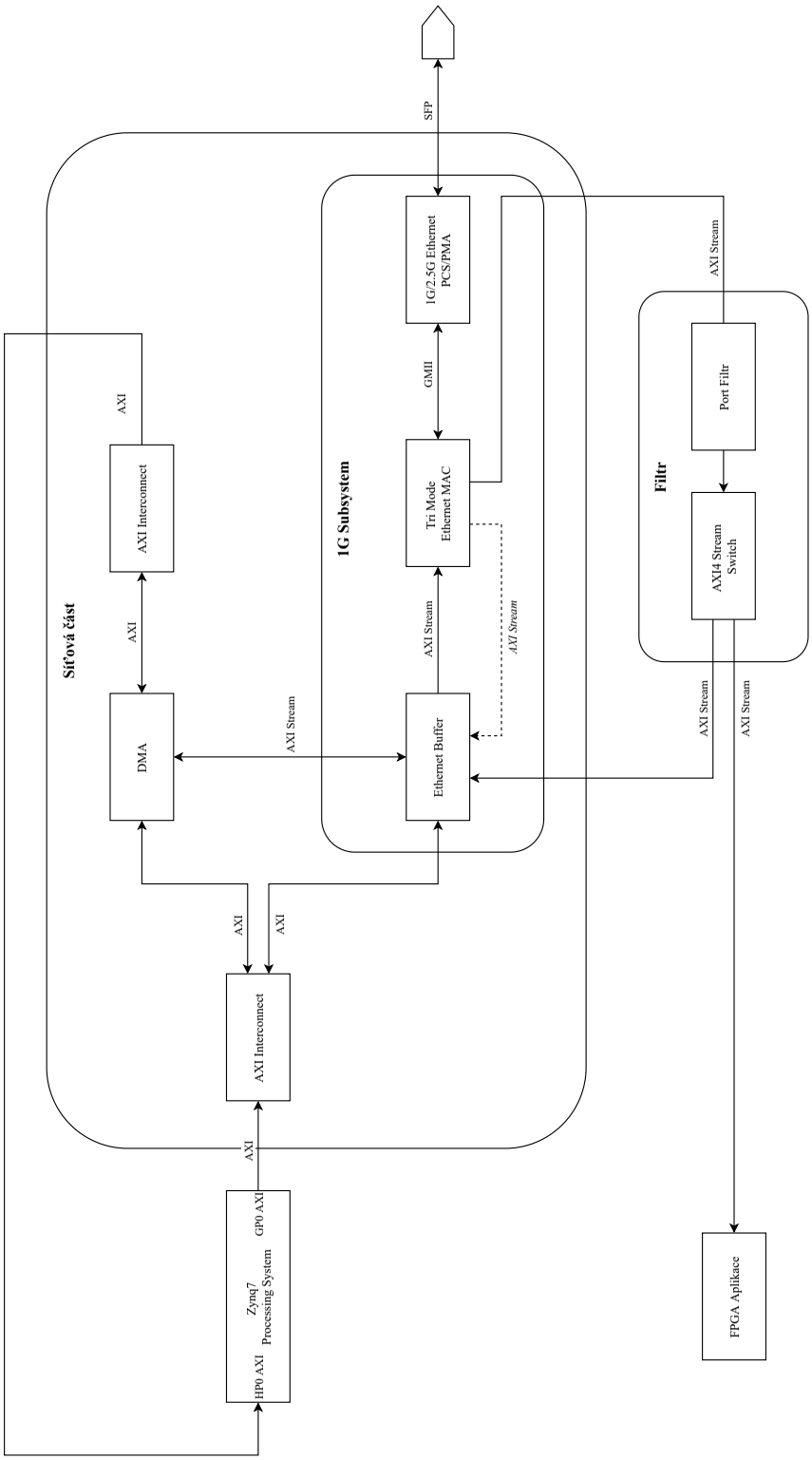
Při spojování dalších modulů jsem vycházel z block designu pro 10G; přidal jsem Zynq7 PS block, mezi síťovou část a Zynq PS jsem vložil DMA, mezi DMA a Zynq PS jsem přidal AXI Interconnect, stejně tak mezi DMA a Ethernet buffer. Také adresy jsem nastavil stejným způsobem.

Konečné zapojení s filtrem je vidět na obrázku 3.3, včetně rozhraní AXI Stream (přerušovaně) uvnitř 1G subsystému, které není dostupné jako externí port. Z tohoto designu bylo možné vytvořit bitstream.

1G verze vyžadovala jiné záznamy v device tree. Ethernet buffer má vlastní typ zařízení a také vyžaduje uvedení typu hodin. Dále bylo nutné přidat údaj k DMA. Tato část device tree je vypsána ve výpisu kódu 3.4.

V OS Petalinux jsem povolil kromě ovladačů zmíněných u 10G také Xilinx AXI DMA Engine.

Po sestavení správně nastavené verze Petalinux systém nabootoval, rozhraní `eth1` se zobrazilo, po nastavení IP adresy, masky sítě a povolení začalo fungovat. Další ověření jsem vypsal v kapitole Testování.



Obrázek 3.3: Zapojení modulů pro rychlost 1G

Uvádím pouze AXI a SFP rozhraní, vícenásobná spojení jsou reprezentována jednou šipkou.

```
&eth_buf{
    local-mac-address = [00 0a 35 00 00 02];
    phy-handle = <&eth_buf_phy>;

    mdio {
        eth_buf_phy: phy@1 {
            compatible = "Xilinx PCS/PMA PHY";
            device_type = "ethernet-phy";
            xlnx,phy-type = <1>;
            clocks = <&jitter_clk>;
            reg = <1>;
        };
    };
};
&axi_dma_0 {
    compatible = "xlnx,eth-dma";
};
```

Výpis kódu 3.4: 1G transceiver node v device tree

3.2.3 Přepínání

Na rozdíl od předpokladů kvůli omezení desky ZC706 v použité verzi IP core od Xilinx není možné dynamicky překonfigurovat připojení k SFP transceiveru za běhu. Je tedy nutné rychlost nastavit již při návrhu pomocí parametru, aby se syntetizovala pouze vybraná část designu.

3.2.3.1 Statické přepínání

Výběr 1G nebo 10G rychlosti je určen hodnotou generic (resp. parameter pro Verilog) `SPEED_IS_10G`. Podle této volby se do designu zahrnou odpovídající bloky.

Do této chvíle jsem implementoval pouze na úrovni block designů, ty ale neumožňují použít generic hodnoty pro podmíněnou syntézu. Navíc cílem práce je vytvořit 1 modul, který lze přidat do již existujících projektů a na různé desky, případně jich použít několik v jednom designu, a je tedy nutné oddělit část obsahující Zynq procesor od zbylé části designu.

V tuto chvíli jsem se rozhodl přesunout celý design na úroveň RTL textu s použitím existujících IP jader z block designu ve formě xci souborů, toto

by umožnilo vložení celého jádra do jiného block designu. Pro tento účel má CESNET vytvořený skript, kterému se předají xci a HDL soubory a on vytvoří Vivado projekt a spustí překlad na bitstream.

Zde nastal problém s IP jádrem Ethernet buffer. Vivado ho nedokázalo načíst pomocí xci souboru, pravděpodobně kvůli nedostupnosti tohoto jádra běžnými způsoby (např. nelze ho nalézt v IP katalogu).

Podařilo se mi Ethernet buffer přidat do projektu jako samostatný block design obsahující pouze dané IP jádro. Toto řešení fungovalo, ale způsobilo, že finální IP jádro nebude možné přidat do block designu kvůli omezení Vivado, že block design nemůže uvnitř obsahovat jiný block design.

Druhým způsobem, jakým se mi podařilo Ethernet buffer přidat, bylo zabalení samotného Ethernet bufferu jako vlastní IP jádro pomocí nástroje ve Vivado. S touto verzí také nebyly další problémy a lze ho přidat do block designu, ale vlastní IP core neumožnuje změnu parametrů bez nového zabalení IP jádra po každé změně.

Při konzultaci s vedoucím jsme se rozhodli pro možnost využívající block design.

Vrátil jsem velkou část projektu do menších oddělených block designů, abych je mohl vhodně spojit pomocí textových souborů obsahující generic. S rozdělenými designy se objevil problém s komunikací mezi procesorem a programovatelnou logikou. Tento problém nastane ve chvíli kdy 1 block design obsahující všechna IP jádra rozdělím na dva (bez přepínání rychlostí). I při identickém nastavení a téměř stejném zapojení (některé signály nelze manuálně přidat, ani s pomocí dalších IP jader) se mi nepodařilo zprovoznit dlouhodobě fungující komunikaci mezi DMA a procesorem na rozhraní AXI HP0.

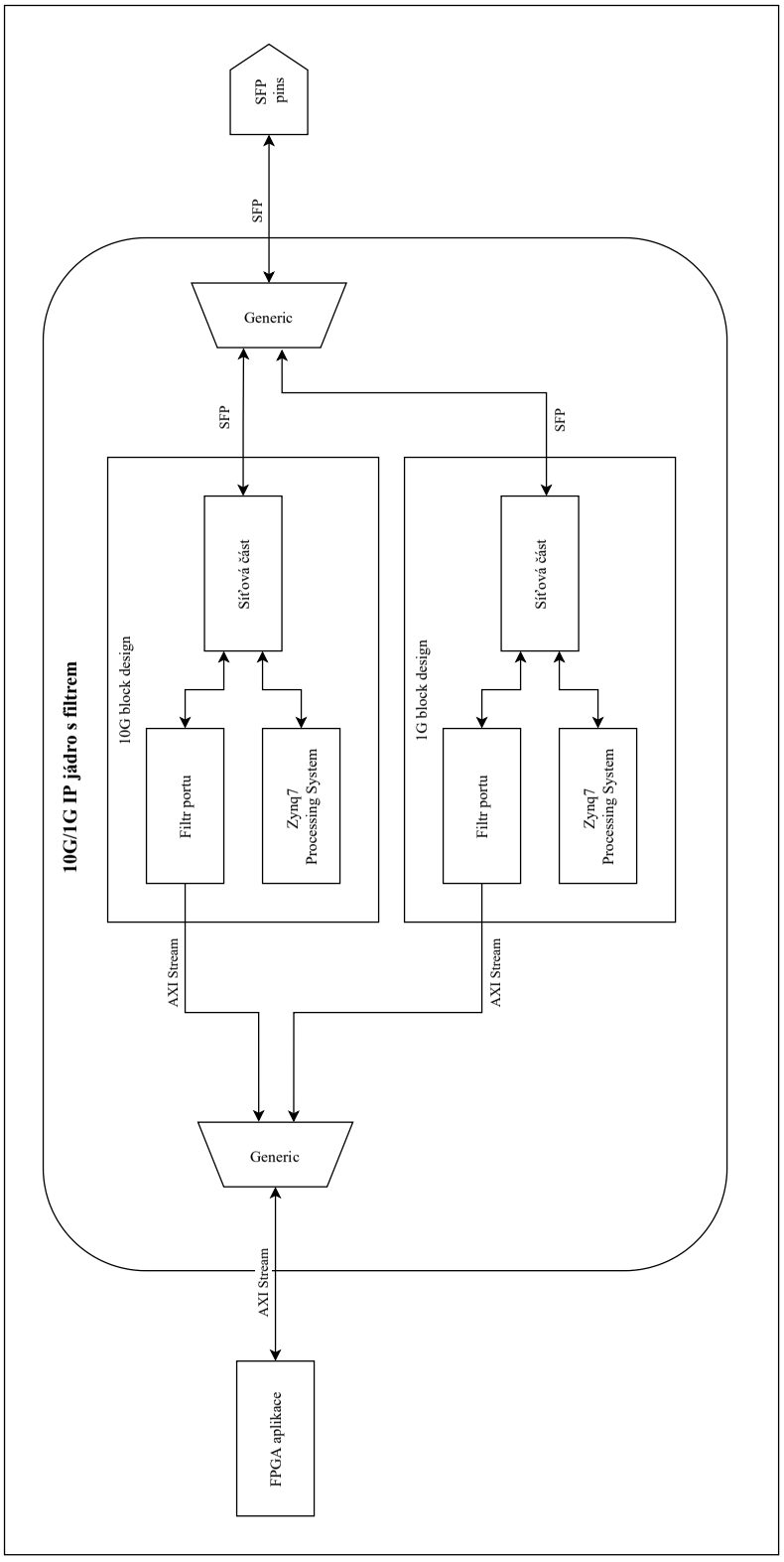
Vzhledem k faktu, že v každém případě v projektu bude block design (tedy nepůjde přidat do jiného BD), jsem se rozhodl použít funkční jednotlivé designy, které obsahují všechna IP jádra spolu. Ty spojím jedním souborem, ve kterém se podle konstanty zvolí, jaký block design se a k němu se připojí všechny vstupní/výstupní signály: AXI Stream rozhraní, 4/5 externích SFP pinů (pro 10G i `TX_disable`), diferenciální hodiny z oscilátoru a statické 200 MHz hodiny pro 1G.

Výsledek je zobrazen na diagramu 3.4. Část `net_1g_10g` je moje výsledné IP jádro, které uvnitř obsahuje 1G a 10G část.

3.2.4 Filtr aplikace

Filtr je umístěn na AXI Stream rozhraní uvnitř části zpracovávající SFP signály, rozděluje část zpracování SFP signálu od zápisu do paměti. Skládá se ze dvou nově vytvořených modulů: Port finder a Dest switch.

Port finder na vstupním rozhraní přijímá proud paketů, na výstupu nezměněné pakety posílá dál, spolu s informací o jejich cílovém portu, pokud je platný protokol. Dest switch data a tyto informace obdrží a podle nich odesílá data na správná rozhraní.



Obrázek 3.4: Zapojení kombinace 10G a 1G pomocí generic

3.2.5 Port Finder

Modul využívá konečný automat pro ovládání signálů, AXI Stream frontu pro zadržení paketu, než se najde cílový port, a několik registrů a kombinačních obvodů pro ověření protokolu a nalezení, uložení a odeslání správného čísla portu.

3.2.5.1 Konečný automat

Základní struktura konečného automatu je vyobrazena na diagramu 3.5, pro přehlednost jsem některé informace vynechal. Výstupy uvedu pouze slovně, jedná se o automat typu Mealy, a pokud nejsou splněny podmínky na uvedených přechodech, zůstává automat ve stejném stavu.

Začíná ve stavu `Zero_Packets` a bez vysílání dat čeká na přijatou transakci. Jakmile ji obdrží, přejde do stavu `One_Packet` a čeká než se najde protokol, poloha a následně číslo cílového portu daného paketu. Poté změní stav na `Sending` a začne vydávat signál odesílat data, stále povoluje příjem nového paketu. Pokud se současný paket odešle dříve, než začne příjem dalšího, automat se přesune zpět do stavu `Zero_Packets` a cyklus se opakuje.

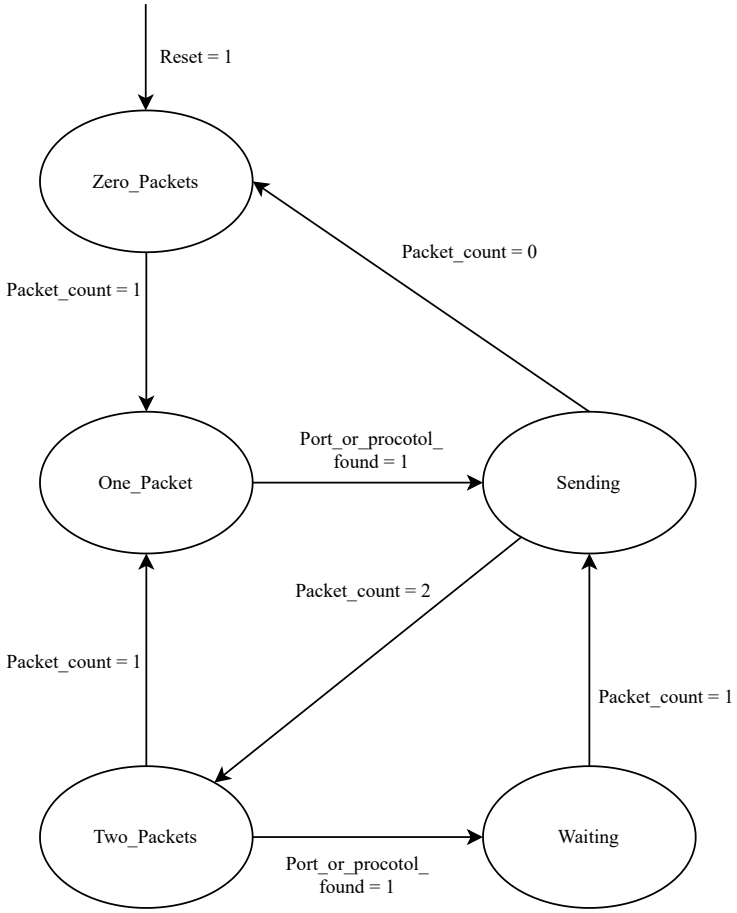
Při vyšším zatížení následující paket přijde ještě před tím, než se první odešle. V takovém případě se přesune do stavu `Two_Packets`, kde čeká na nalezení a načtení protokolu a portu druhého paketu do záložních registrů a zároveň kontroluje dokončení odeslání. Pokud se odešle, vyšle signál, aby se načetla hodnota ze záložních registrů a pokračuje se ze stavu `One_Packet`. V případě, kdy má automat nalezený port i druhého paketu, zastaví přijímání AXI Stream transakcí a čeká na dokončení odeslání dat ve stavu `Waiting`.

3.2.5.2 Kombinační obvody a registry

Pro ovládání přijímání a odesílání dat jsem napojil signály z automatu přímo na vstupní/výstupní signály a porty fronty TREADY a TVALID pomocí hradla AND, tak aby všechny moduly dostávali správné informace.

Informaci o cílovém portu posílám na port TDEST výstupního AXI rozhraní.

Než mohu získat hodnotu portu, musím nejdříve z IP hlavičky vyčíst její celkovou délku. Tento údaj se nachází v prvním bytu IP hlavičky, která následuje po Ethernet hlavičce o pevné délce 14 bytů. Abych věděl jaké byty



Obrázek 3.5: Základní struktura konečného automat entity Port finder

jsou právě přijímány z AXI Stream rozhraní, přidal jsem registr `byte_cnt` jako počítadlo přijatých bytů, které se přizpůsobí zadané šířce datového signálu. Podle `byte_cnt` určím, při jaké transakci a z kterých bytů vyčtu délku hlavičky, potom vypočítám v jakém bytu se nachází cílový port.

Následně najdu pole, které obsahuje informaci o protokolu paketu, se speciální verzí pro šířku dat 8 bitů, protože délka čísla protokolu je 16 bitů, a musí se tedy načíst ve dvou cyklech. To provádím jednoduše tak, že v prvním taktu uložím data obsahující jeden byte dat, a v dalším taktu nová data na vstupu a uložená data vystavím na výstup TDEST.

Hodnotu porovnám s pevně zadanými konstantami v souboru (například 6 a 17 pro TCP a UDP resp.) a podle ní nastavím `protocol_ok` pro současný paket na 1 (protokol odpovídá, na TDEST pošlu číslo portu) nebo 0 (jiný než povolený protokol, TDEST má hodnotu konstanty `DEST_PROTO_INVALID`). V případě 0 se také automatu oznámí, že se může odesílat paket, v opačném případě se toto pošle až se nalezne hodnota cílového portu. Ta se načítá podobným způsobem, má šířku 8 bitů a tedy nevyžaduje speciální přístup v případě šířky dat 8 bitů.

Hodnoty protokolu a následně i portu uložím ho do primárního nebo sekundárního registru podle signálu z automatu.

Další důležitá interní hodnota je `packet_cnt`. Ta určuje počet aktuálně přijatých paketů v modulu. Snížení hodnoty se určuje jednoduše, děje se při potvrzení odeslání transakce s last nastaveným na 1. Zvýšení funguje podobně, při obdržení transakce s aktivním last se nastaví signál `packet_inc`. Pomocí těchto signálů se v každém taktu do registru uloží správná hodnota `packet_cnt`.

3.2.5.3 AXI Stream fronta

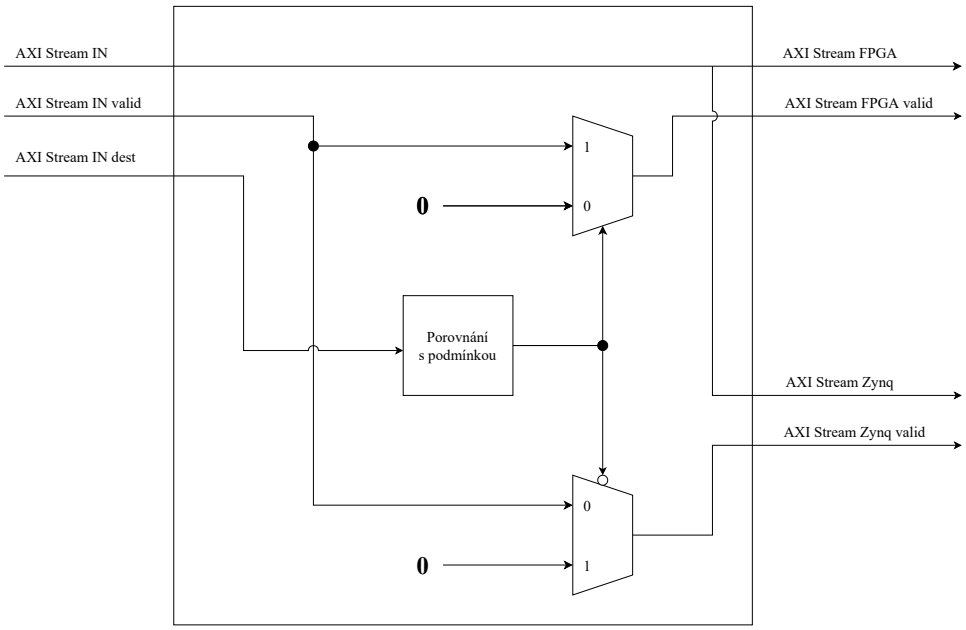
Instance fronty je jednoduchá. Většinu portů jsem spojil přímo se vstupem a výstupem. Pouze na TREADY a TVALID jsem přivedl vlastní signály, ovládané automatem.

Automat funguje maximálně se 2 pakety o předem nastavené velikost, frontu jsem nastavil na dvojnásobek, aby existovala rezerva. Lze ji snadno změnit hodnotou generic. Toto řešení minimalizuje velikost a náročnost samotného modulu Port finder, ale pokud není výstupní rozhraní přijímat všechna data, vyžaduje vhodnou frontu před vstupním AXI Stream rozhraním.

3.2.6 AXI Dest Switch

Pro rozdělení paketů do 2 různých AXI Streamů jsem nejdříve plánoval použít IP code AXI Stream Switch od Xilinx. To rozděluje AXI transakce ze slave na master rozhraní podle hodnoty signálu TDEST. Při nastavování IP jádra jsem zjistil, že možnosti směrování jsou poměrně omezené. Ke každému portu lze přiřadit pouze jeden rozsah adres, což v případě portů, které mají často nesouvislé hodnoty, není vhodné.

Rozhodl jsem se vytvořit vlastní modul pro směrování AXI Streamů. Data posílám na obě výstupní rozhraní. Pro odesílání na správné rozhraní musím měnit pouze TVALID, které je nastaveno na 0 na rozhraní, kterému nejsou data určena, na druhém rozhraní nechávám tento signál procházet volně. Podobně ovládám kontrolní signál TREADY, který propouštím podle hodnoty DEST. Řešení je vyobrazeno na diagramu 3.6.



Obrázek 3.6: Modul Destination Switch

AXI Stream nezahrnuje TREADY, TVALID ani TDEST, jsou zobrazeny odděleně.

K určení, kam data poslat, stačí jednoduchá podmínka. Zde moje řešení nabízí mnohem větší svobodu než zmíněné IP jádro. Porovnávací podmínka může být jakákoliv, za předpokladu, že půjde syntetizovat na FPGA. V mém případě jsem se rozhodl pro rovnost se dvěma hodnotami, které lze nastavit pomocí generic.

Tato implementace je velmi nenáročná na zdroje – za předpokladu, že porovnávací podmínka nevyžaduje funkce sekvenčního obvodu, tento modul funguje jako prostý kombinační obvod. V takovém případě nepotřebuje přivedené hodiny ani reset.

3.2.6.1 Spojení vlastních modulů

Poslední dva zmíněné moduly, Port Finder a AXI Stream Dest Switch, jsem zabalil v jednoduchém bloku `port_filter`.

Finální modul funguje přesně, jak je požadováno – na jediném vstupním AXI Stream rozhraní přebírá data v podobě Ethernet rámců, které obsahují pakety Internet protokolu, a na 2 výstupních AXI Stream rozhraních posílání stejná data, rozdělená podle cílové portu.

3.3 Spojení do IP jádra

Do projektu jsem přidal soubory AXIStreamPortFilter, AXIStreamPortFinder, AXIStreamDestSwitch a AXIStreamFIFO. Zapojení jsem provedl v obou block designech identicky – do designu jsem vložil jsem RTL modul filtru portu a dvě AXI Stream fronty – jednu před filtr, jednu na stranu FPGA jako reprezentaci aplikace, které jsem konstantou nastavil vstupní TREADY (na výstupním AXI Stream) permanentně na 1, aby mohl provoz neustále procházet. Změnil jsem VHDL verzi souboru AXIStreamPortFinder na VHDL-2008 a správně nastavil rychlost hodinového signálu a jeho hodinovou doménu u AXI rozhraní v novém modulu. Poté bylo možné design syntetizovat a vytvořit z něj bitstream.

Kapitola 4

Testování a verifikace

V této kapitole popisuji jakým způsobem jsem ověřoval správnost mého návrhu. Nejdříve jsem mnou vytvořené moduly simuloval, poté jsem je testoval na desce v reálném provozu.

4.1 Simulace

Pro simulaci jsem využil dvě verze, jednu pro kontrolu správného chování s nestálým vstupním AXI rozhraním, druhou pro ověření korektního zpracování dat.

4.1.1 Jednoduchá simulace

Nejdříve jsem Port Filter simuloval s pomocí jednoduchého modulu, který jsem vytvořil a pojmenoval `AXIS_Send`. Ten postupně posílá vstupní data na výstupní AXI Stream rozhraní. Vstupem je libovolně velký vektor obsahující všechna data při startu simulace, tyto data jsou konstantně definovaná ve zdrojovém souboru simulace. Signál `AXI_TVALID` je pomocí náhodné funkce nastaven na 1 nebo 0 při každém hodinovém cyklu, ovšem dodržuje AXI protokol – pokud `AXI_TVALID` byl v posledním taktu nastaven na hodnotu 1 a transakce neproběhla, zůstává na 1. Tímto jsem ověřil zpracování dat při různých kombinacích TVALID a TREADY. Jakmile testování proběhlo v pořádku, pokračoval jsem ověřením správného zpracování a odesílání dat.

Dále už se správné chování při nestabilním signálu TVALID vstupního AXI Stream rozhraní neověřuje, PCAP master má během přenosu jednoho

paketu TVALID stále nastaven na 1. Na druhou stranu přesné dodržení AXI protokolu na výstupním rozhraní kontroluje verifikační IP.

4.1.2 Scoreboard simulace

Vytvořil jsem testbench zahrnující scoreboard podle ověřených designů, vyobrazen na diagramu 4.1. Jako vstup lze použít libovolný PCAP soubor obsahující Ethernet rámce s IP pakety uvnitř. Pro otestování `port_filter` je vhodné v souboru zahrnout protokoly TCP nebo UDP s cílovými porty 54238 a 1900 – tyto hodnoty platí ve výchozím stavu, lze samozřejmě změnit pomocí argumentu pro `split-traffic.sh` a generic v hlavním souboru simulace `AXIStreamPortFilterSim.sv`. Protože můj modul má dvě výstupní AXI Stream rozhraní, obsahuje simulace dva porovnávací procesy a dva referenční vstupy.

Skript `split-traffic.sh` jako argumenty očekává PCAP soubor (v mém příkladu `full.pcap`), a čísla cílových portů, podle kterých oddělí pakety určené aplikaci v FPGA do `fpga.pcap`, zbylé uloží do `zynq.pcap`. Tyto tři soubory se poté použijí jako zdroj dat pro testbench.

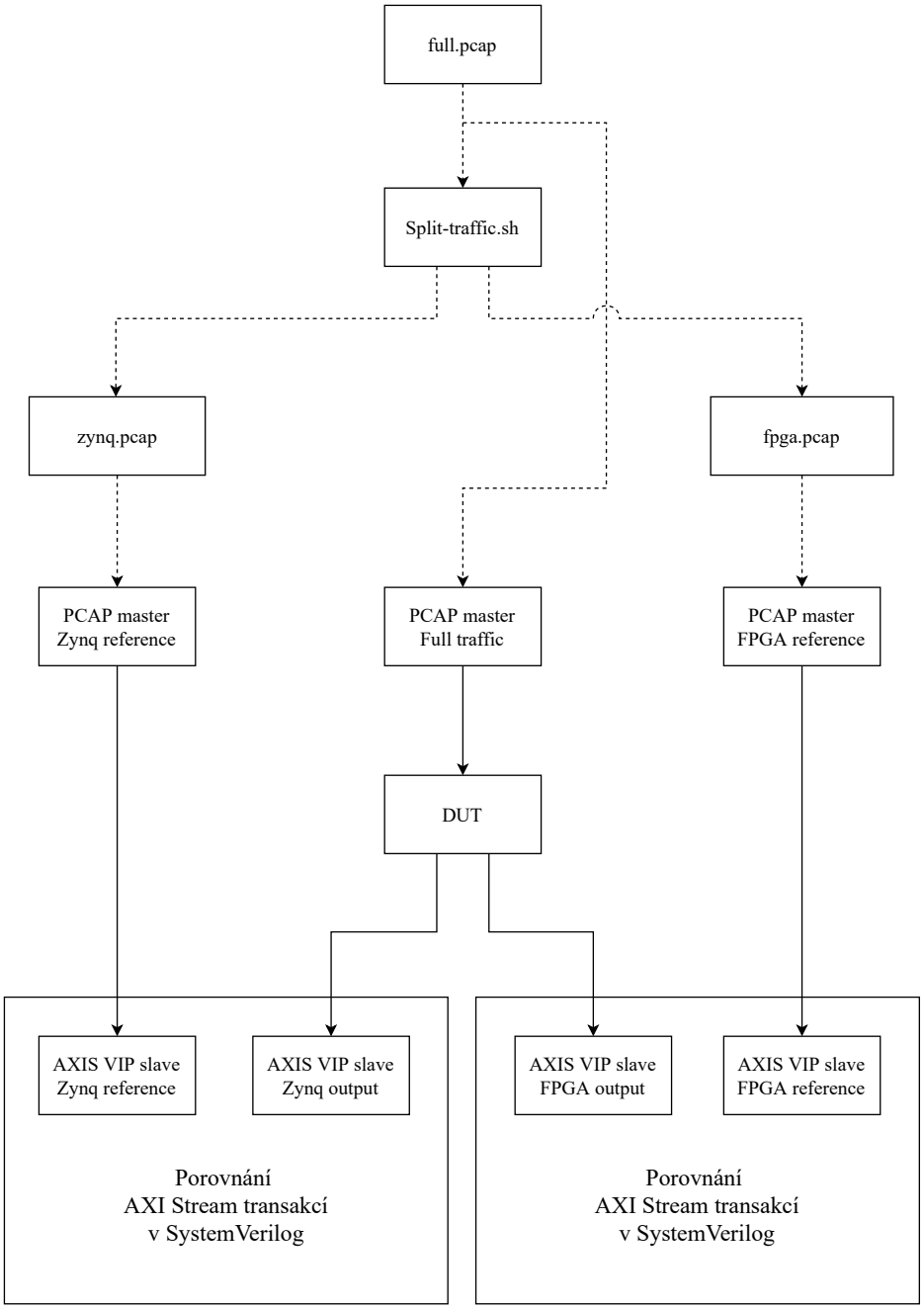
HDL simulace obsahuje 3 instance PCAP master, které vytvářejí AXI Stream transakce z pcap souborů, které jsou nastaveny parametrem v souboru simulace `AXIStreamPortFilterSim.sv`. Jedna instance slouží jako zdroj dat pro testovaný modul (DUT, device under test), zbylé dvě posílají data instancím VIP (verifikační IP) slave, které slouží jako reference.

DUT podle specifikací přijme AXI Stream pakety, rozdělí je podle určených cílových portů a odešle je na správné rozhraní. Každé z nich je zapojeno na VIP slave, které zpracuje a zkontroluje transakci a data uloží.

VIP jsem implementoval podle [37]. Ve vrchním modulu simulace jsou permanentně spuštěny 2 scoreboard procesy, které tato data nepřetržitě porovnávají s daty z referenčních VIP slave. V případě, že se data neshodují, vypíše se chyba na výstup konzole, pokud jsou data stejná, nevypíše se nic. Počty chyb také ukládají do paměti.

Na konci simulace (určené konstantou odpovídající počtu nanosekund simulace) se vypíše zpráva o počtu chybných a celkovém počtu transakcí.

Otestoval jsem přiložené sady dat pro šířky dat postupně 64, 32, 16 a 8 bitů. Do 16 bitů nebyl se simulací problém. Šířka 8 bitů potřebuje jiný přístup, protože hodnota cílového portu je reprezentována 16 bity. S tím se v simulaci



Obrázek 4.1: Struktura simulace

Přerušované spojení označuje práci se soubory, plné AXI Stream.

objevil problém. Přes to, že simulační prostředí jsem použil identické, v hodinovém cyklu, kdy vstupní data obsahovala port, se nepropagovala na výstupní datové signály.

Část speciálně modifikovanou pro 8 bitový vstup jsem pozměnil, aby si ukládala všech 16 bitů do registru a na výstup je vystavovala až v dalším hodinovém cyklu. To by v designu způsobilo zpoždění o jeden hodinový cyklus, což se ale pohybuje v řádu jednotek nanosekund a je naprosto zanedbatelné.

Po této změně fungovala simulace všech uvedených šířek bez problému. Moje moduly požadují šířku alespoň 8 bitů a hodnota musí být mocnina 2. Síťová IP jádra v této práci mají šířku datového signálu 8 a 64 bitů, takže není cílem podporovat jiné velikosti. Kompatibilitu se šířkami 16 a 32 bitů zajišťuje moje řešení přirozeně. Implementaci by bylo možné rozšířit, aby fungovala pro libovolný počet bytů, ale takové šířky nebývají běžné a řešení by mohlo zbytečně plýtvat zdroji, pokud takové hodnoty nejsou v designu nikdy použity.

4.2 Testování

Dále jsem testoval mnou vytvořený design nahraný na desku ZC706 v reálném provozu, který jsem posílal z počítače. Přípojen byl buď přímo (point-to-point) nebo přes switch.

4.2.1 Způsob testování běžného provozu

Požadavkem pro další testování je úspěšné povolení síťového rozhraní příkazem `ifconfig up`, s nastavenou IP adresou a maskou. To demonstruje schopnost OS a ovladače komunikovat s IP jádrem. Tento krok a jeho úspěch byl již zmíněn v kapitole implementace u každé z verzí.

Dále jsem použil základní nástroj pro testování připojení – `ping`. Tento jednoduchý program odešle ICMP dotaz, na který cílová stanice odpoví ICMP odpovědí. Proces ověří schopnost zjistit vzdálenou MAC adresu (odeslat broadcast, zpracovat ARP pro zjištění cílové MAC adresy), odeslat data do sítě a přijmout data.

Pomocí `ping` nemusí být možné testovat složitější systémy, které využívají firewall a mohou přímo zakázat ICMP odpovědi. Systém použitý v této práci je však jednoduchý, bez vlastnoručně nastaveného firewallu, ping tedy

podává odpovídající představu o konektivitě. Ping neklade přísné nároky na načasování odpovědi, dovoluje cíli odpovědět až po několika sekundách.

Na závěr testování běžné síťové komunikace jsem se zkusil připojit k desce přes SSH. Komunikační protokol používá TCP a šifrování, což otestuje schopnost přijmout, zpracovat a odpovědět na každý paket beze ztráty. Testoval jsem ho také proto, že SSH je běžné při práci se síťovými zařízeními, často k nim není snadný fyzický přístup a snadno lze měnit cíl připojení.

4.2.1.1 Verze 10G

Po nastavení a povolení rozhraní připojení přes 10G fungovalo bez problému, deska odpověděla na ping, včetně získání MAC adresy pomocí ARP. Stejně tak bylo možné se k běžícímu OS Petalinux připojit přes SSH.

4.2.1.2 Verze 1G

1G vyžadovala komplexnější manuální nastavení, které s sebou přinesly některé problémy. Nejdříve nesprávné použití AXI Stream fronty, kterou jsem bez problému použil v 10G designu, vedlo k opoždění paketů. Právě v tomto případě ping ukazoval konektivitu, ale se zpožděním v řádu sekund – právě tolik času kolik trvalo naplnit frontu dalšími ICMP požadavky. Chyba byla pouze na straně přijímání dat, takže odeslání fungovalo správně a uskutečnilo se okamžitě.

Po odstranění AXI Stream front se nejdříve zdálo, že problém přetrvává v menším měřítku – všechny přijaté pakety jsou opožděny o jeden paket. Jednoduchý příkaz ping toto chování interpretuje jako délku zpoždění 1 sekunda. To je právě doba, po níž se pošle další ICMP paket a původní paket je předán ovladači v OS, který konečně vyšle odpověď. Takovéto chování znemožnilo fungování SSH, protože nebylo možné navázat zabezpečené spojení. Žádná fronta však v designu nebyla.

Po dalším testování jsem zjistil, že po manuálním vypnutí a zapnutí rozhraní `eth1` vše začalo fungovat bez problému. ICMP odpovědi začali přicházet do 1 milisekundy a bylo možné bez problémů navázat SSH spojení.

4.2.2 Testování filtru

Správné chování filtru na desce jsem ověřil opět pomocí nástrojů ping a SSH, spolu s mnohem pečlivějším monitorováním přes programy Wireshark na Windows PC a tcpdump na desce ZC706. Nejdříve jsem zkusil jednoduchý ping, poté připojení SSH. Jakmile fungovalo, měnil jsem jeho port a kontroloval přijatý provoz na každém zařízení.

4.2.2.1 10G

Začal jsem s 10G, protože fungovalo dříve, bezproblémově a navíc šířka dat AXI Stream rozhraní je 64 bitů, pro což jsem navrhoval filtr portu od začátku. Použil jsem design z předchozí kapitoly.

ICMP požadavky i odpovědi procházeli od PC k ARM procesoru a OS Petalinux i zpět. Stejně tak bylo možné se připojit a ovládat OS přes SSH. Tomuto chování odpovídal i zachycený provoz – nejdříve ARP dotaz, odpověď na něj, poté běžná oboustranná komunikace. Když jsem změnil port připojení na jiný a pokusil se připojit z PC, k desce se nepřipojil, ale bylo na ní vidět několik paketů poslaných oběma směry na obou zařízeních.

Jakmile jsem změnil port na ten nastavený ve filtru (1234, 12345), Wireshark na testovacím PC stále zobrazoval odeslané pakety, ale žádné odpovědi. Na desku z pohledu OS jak podle tcpdump, tak ifconfig nepřišly žádné pakety (dokud PC neposlalo ARP žádost, protože cíl neodpovídal), což přesně odpovídá požadovanému chování.

4.2.3 1G

Komponenty fyzické vrstvy pro zpracování 1G signálu používají šířku dat 8 bitů. To vyžaduje mírně odlišné chování od 10G verze.

Základní komunikace fungovala bez problému: ping, SSH ve výchozím nastavení i SSH na různé porty.

Filtr ale nefungoval – propouštěl přijaté pakety k procesoru i přes to, že paket s tímto cílovým portem měl zůstat v programovatelné logice, stejně jako v případě 10G. Přidal jsem proto ILA (integrovaný logický analyzátor) a zjistil, že signál na TDEST, který se má rovnat portu byl o jeden byte opožděn, resp. že se načítá o jeden hodinový takt později. To přesně odpovídalo mým prvotním očekáváním, ale v simulaci takový kód nefunguje a načítání musí

být zpožděno o jeden hodinový cyklus. Soubory pro implementaci a simulaci jsem tedy rozdělil.

Po snadné opravě IP jádro fungovalo podle požadavků, se stejnou chybou jako 1G bez filtru – po bootu OS bylo nutné rozhraní eth1 nejdříve zapnout, vypnout a znovu zapnout, jinak byly pakety přijímány se zpožděním.

Dále test probíhal stejně jako v případě 10G. ICMP pakety přicházely okamžitě na obě zařízení, SSH bez parametrů vytvořilo připojení, přes které jsem mohl ovládat OS na desce. SSH na jiné (než 22 a filtrované) porty poslalo žádost o připojení, kterou zaznamenal OS a bylo vidět v tcpdump spuštěném na desce. Naproti tomu pokus o komunikaci na portu 1234 nebo 12345 (porty nastavené na filtru) stejně jako u 10G zaznamenal Wireshark na PC, ale do OS Petalinux na desce nedorazil ani jeden paket s těmito cílovými porty.

4.3 Výkon

Maximální rychlost komunikace jsem netestoval. Tato deska z tomu není přímo určena, nebylo to účelem práce a již se to nevešlo do rozsahu práce.

Dosažení téměř rychlostí 1 gigabit/s a 10 gigabit/s by neměl být problém, IP jádra pro zpracování signálu samozřejmě dokáží komunikovat takto rychle a moje vytvořené moduly pracují na stejných frekvencích (125 MHz a 156,25 MHz resp.) bez problémů s časováním. Maximální zpomalení může být 1 hodinový cyklus na paket, což by mohlo být znatelné u běžných paketů, např. u 54 bytů to znamená ztrátu rychlosti 1/7, ale takové pakety nejspíše nezatíží ani 1G připojení. Pakety, které přenášejí velké objemy dat, budou nejspíše značně delší, běžně mají délku 1500 bytů a v takovém případě by se jednalo o teoretické snížení rychlosti o 1/187, což se reálně neprojeví.

Závěr

Cílem práce bylo vytvořit IP jádro, které se připojí na SFP port na desce ZC706 a zpracovává síťový provoz z vloženého transceiveru a podle portu a odesílá buď na AXI Stream rozhraní nebo do OS.

To se mi z větší části podařilo, 10G řešení funguje podle očekávání. 1G řešení má menší nedostatky, bohužel nebyly vyřešeny v rámci časového rámce práce. Naštěstí nedostatek lze vyřešit softwarovým řešením v podobě restartu ethernetového rozhraní po prvním použití při inicializaci linuxu. Po aplikování restartu 1G řešení pracuje korektně v otestovaném rozsahu.

Vytvořené IP jádro provoz na úrovni Ethernet rámců filtruje podle cílového portu, pakety s určenými cílovými porty zachytí a pošle na AXI Stream rozhraní určené pro filtrovaný provoz. Zbytek paketů přeposílá do paměti procesoru, kde si je převezme správně nastavený ovladač v OS Petalinux.

Kvůli omezením od výrobce nelze na použité vývojové desce mezi rychlostmi přepínat za běhu, vybranou rychlost 1G nebo 10G je nutné nastavit již při návrhu. Toto omezení nebylo na první pohled zjevné, protože výrobce uváděl tuto vlastnost v podobě aplikační poznámky. Později se ukázalo, že výrobce v novějších verzích vývojových nástrojů podporu odebral.

Tato restrikce na novější deskách s FPGA Ultrascale není, budoucí práce by mohly implementovat dynamické přepínání. Další možné vylepšení by mohlo pozměnit implementaci 1G tak, aby podporovala libovolné fyzické rozhraní pomocí MII namísto SFP a přidat ji jako další možnost.

Bibliografie

1. ZIMMERMANN, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. 1980, roč. 28, č. 4, s. 425–432. Dostupné z DOI: 10.1109/TCOM.1980.1094702.
2. PETERKA, Jiří. *Co je čím …v počítačových sítích – Fyzická vrstva* [online]. 1992 [cit. 2021-06-16]. Dostupné z: https://www.earchiv.cz/a92/a217c110.php3.
3. PETERKA, Jiří. *Co je čím …v počítačových sítích – Linková vrstva - I.* [online]. 1992 [cit. 2021-06-16]. Dostupné z: https://www.earchiv.cz/a92/a218c110.php3.
4. ASSOCIATION, IEEE Standards. *Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)* [online]. 2017 [cit. 2021-06-16]. Dostupné z: https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf.
5. PETERKA, Jiří. *Co je čím …v počítačových sítích – Síťová vrstva - I.* [online]. 1992 [cit. 2021-06-26]. Dostupné z: https://www.earchiv.cz/a92/a221c110.php3.
6. INFORMATION SCIENCES INSTITUTE, UNIVERSITY OF SOUTHERN CALIFORNIA. *Transmission Control Protocol* [online]. 1981 [cit. 2021-06-26]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc793.

7. POSTEL, J. *User Datagram Protocol* [online]. 1980 [cit. 2021-06-26]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc768.

8. *IEEE 802.3 ETHERNET WORKING GROUP* [online]. 2021 [cit. 2021-06-20]. Dostupné z: https://www.ieee802.org/3/.

9. NORIVAL FIGUEIRA, et al. *10GE WAN PHY:Physical Medium Attachment (PMA)* [online]. 2000 [cit. 2021-06-20]. Dostupné z: https://grouper.ieee.org/groups/802/3/ae/public/mar00/figueira_1_0300.pdf.

10. SFF COMMITTEE. *SFP (Small Formfactor Pluggable) Transceiver* [online]. 2001 [cit. 2021-06-26]. Dostupné z: https://www.10gtek.com/templates/wzten/pdf/INF-8074.pdf.

11. TÉNART, Antoine. *SFP modules on a board running Linux* [online]. 2020 [cit. 2021-06-23]. Dostupné z: https://bootlin.com/blog/sfp-modules-on-a-board-running-linux/.

12. BIONDI, Philippe. *Scapy* [online]. 2021 [cit. 2021-06-15]. Dostupné z: https://scapy.net/.

13. XILINX. *Virtex UltraScale* [online]. 2021 [cit. 2021-06-15]. Dostupné z: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html.

14. IEEE Standard for Ethernet – Amendment 7: Physical Layer and Management Parameters for 400 Gb/s over Multimode Fiber. *IEEE Std 802.3cm-2020 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018, IEEE Std 802.3bt-2018, IEEE Std 802.3cd-2018, IEEE Std 802.3cn-2019, IEEE Std 802.3cg-2019, and IEEE Std 802.3cq-2020)*. 2020, s. 1–72. Dostupné z DOI: 10.1109/IEEESTD.2020.9052826.

15. XILINX. *AXI 1G/2.5G Ethernet Subsystem v7.0 Product Guide* [online]. 2017 [cit. 2021-06-15]. Dostupné z: https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf.

16. XILINX. *10 Gigabit Ethernet Subsystem v3.1 Product Guide* [online]. 2021 [cit. 2021-06-15]. Dostupné z: https://www.xilinx.com/support/documentation/ip_documentation/axi_10g_ethernet/v3_1/pg157-axi-10g-ethernet.pdf.

17. MAXFIELD, Clive. *Fundamentals of FPGAs: What Are FPGAs and Why Are They Needed?* [online]. 2019 [cit. 2021-06-16]. Dostupné z: https://www.digikey.com/en/articles/fundamentals-of-fpgas-what-are-fpgas-and-why-are-they-needed.

18. SERVER, HW. *Co je FPGA a proč je použít?* [online]. 2019 [cit. 2021-06-16]. Dostupné z: https://vyvoj.hw.cz/zaklady-fpga-co-je-fpga-a-proc-je-pouzit.html.

19. FALLAHLALEHZARI, Farhad. *Processing With FPGAs On Mars* [online]. 2021 [cit. 2021-06-16]. Dostupné z: https://semiengineering.com/processing-with-fpgas-on-mars/.

20. LEDIN, Jim. *Embedded design with FPGAs: Development process* [online]. 2021 [cit. 2021-06-20]. Dostupné z: https://www.embedded.com/embedded-design-with-fpgas-development-process/.

21. LEDIN, Jim. *Architecting High-Performance Embedded Systems: Design and build high-performance real-time digital systems based on FPGAs and custom circuits*. Packt Publishing, 2021.

22. XILINX. *AXI Reference Guide* [online]. 2011 [cit. 2021-06-27]. Dostupné z: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

23. VÕSANDI, Lauri. *Arbitrary data streams* [online]. 2016 [cit. 2021-06-27]. Dostupné z: https://lauri.võsandi.com/hdl/zynq/axi-stream.html.

24. ABHIJIT ATHAVALE, Carl Christensen. *High-Speed Serial I/O Made Simple*. Xilinx, 2005. Dostupné také z: https://www.xilinx.com/publications/archives/books/serialio.pdf.

25. XILINX. *7 Series FPGAs GTX/GTH Transceivers* [online]. 2018 [cit. 2021-06-20]. Dostupné z: www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf.

26. XILINX. *Zynq-7000 SoC* [online]. 2021 [cit. 2021-06-15]. Dostupné z: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

27. SILICON LABORATORIES. *ANY-FREQUENCY PRECISION CLOCK MULTIPLIER/JITTER ATTENUATOR* [online]. 2014 [cit. 2021-06-20]. Dostupné z: https://www.silabs.com/documents/public/data-sheets/Si5324.pdf.

28. XILINX. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide* [online]. 2019 [cit. 2021-06-26]. Dostupné z: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf.

29. ADAM DUNKELS, Leon Woestenberg. *lwIP Lightweight IP stack* [online]. 2021 [cit. 2021-06-15]. Dostupné z: https://www.nongnu.org/lwip/2_1_x/index.html.

30. SAVIN, Dmitriy. *Linux Device Drivers: Tutorial for Linux Driver Development* [online]. 2021 [cit. 2021-06-26]. Dostupné z: https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os.

31. CORBET, Jonathan; RUBINI, Alessandro; KROAH-HARTMAN, Greg. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005. ISBN 9780596005900. Dostupné také z: https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch01.html.

32. LIKELY, Grant. *Linux and the Devicetree* [online]. 2021 [cit. 2021-06-26]. Dostupné z: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html.

33. *CCF SI5324 Driver* [online]. 2020 [cit. 2021-06-27]. Dostupné z: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841874/CCF+SI5324+Driver.

34. WIKI, Xilinx. *Linux I2C Driver* [online]. 2021 [cit. 2021-06-20]. Dostupné z: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841974/Linux+I2C+Driver.

35. REAL-TIME SYSTEMS RESEARCH GROUP AT THE UNIVERSITY OF YORK. *Zynq Ten-Gigabit Example* [online]. 2015 [cit. 2021-06-26]. Dostupné z: https://github.com/RTSYork/zc706_10g_example.

36. *Ethernet Performance with Jumbo Frame Support & PL Ethernet in Zynq-7000 AP SoC* [online]. 2021 [cit. 2021-06-27]. Dostupné z: https://www.linuxsecrets.com/xilinx/Zynq+PL+Ethernet.html.

37. XILINX. *AXI4-Stream Verification IP v1.0* [online]. 2017 [cit. 2021-06-18]. Dostupné z: `https://www.xilinx.com/support/documentation/ip_documentation/axi4stream_vip/v1_0/pg277-axi4stream-vip.pdf`.

Dodatek A

Seznam použitých zkratek

10G Rychlost 10 gigabit/s, 10 Gbit/s

1G Rychlost 1 gigabit/s, 1 Gbit/s

ASIC Application specific integrated circuit, zákaznický integrovaný obvod

AXI Advanced extensible interface

BSP Board support package

CPU Central processing unit, procesor

DHCP Dynamic host configuration protocol

DMA Direct memory access

DUT Device under test, testované zařízení

FIFO First in, first out; fronta

FPGA Field-programmable gate array, programovatelné hradlové pole

FS File system, souborový systém

HDL Hardware description language

I^{2}C Inter-Integrated Circuit

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers, Institut pro elektrotechnické a elektronické inženýrství

IP Intelectual property

IP Internet protocol

ISO International organization for standardization, Mezinárodní organizace pro normalizaci

LLC Logical link control

MAC Media access control

MGT Multi-gigabit transceiver

OSI Open systems interconnection

OS Operační systém

PCAP Packet capturing, zachytávání paketů

PC Personal computer, osobní počítač

PCS Physical coding sublayer

PMA Physical medium attachment

RAM Random access memory

RTL Register-transfer level

RTOS Real time operating system, operační systém reálného času

SFP Small form-factor pluggable

SoC System on chip, systém na čipu

TCP Transmission control protocol

UDP User datagram protocol

VIP Verification IP, verifikační IP

Dodatek B

Obsah přiloženého CD

- `readme.txt`
- `bitstreams` ..... bitstreamy pro nahrání do zařízení
- `simulation` ..... složka obsahující vše pro simulaci
- `source` ..... zdrojové kódy
  - `implementation` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `DP-Brokes-Jan-LS2021.pdf` ..... text práce ve formátu PDF