VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVÁ AKCELERACE EXTRAKCE A SPOJOVÁNÍ POLOŽEK Z HLAVIČEK PAKETŮ

HARDWARE ACCELERATION OF EXTRACTION AND MERGING OF ITEMS FROM PACKET HEADERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MIKULÁŠ BRÁZDA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Brázda Mikuláš**
Program: Informační technologie
Název: **Hardwarová akcelerace extrakce a spojování položek z hlaviček paketů**
Hardware Acceleration of Extraction and Merging of Items from Packet Headers
Kategorie: Návrh číslicových systémů

Zadání:

1. Seznamte se s problematikou extrakce a spojování položek z hlaviček paketů a existujícími přístupy pro hardwarovou akceleraci těchto operací.
2. Seznamte se s existujícími technikami a nástroji pro syntézu obvodů z vyšších programovacích jazyků (HLS).
3. Navrhněte vhodný způsob akcelerace úlohy extrakce a spojování položek z hlaviček paketů s využitím dostupných HLS nástrojů.
4. Proveďte implementaci navrženého řešení a jeho funkčnost ověřte na dostupném hardware.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího práce.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

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

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

Abstrakt

Téměř každé zařízení v síti potřebuje pro svoji činnost vyextrahovat některá pole z hlaviček paketů, provést nad nimi operace a znovu složený paket přeposlat dál. Toto zpracování musí být implementováno na rychlosti odpovídající rychlosti linky. Na vysokorychlostních sítích se pro splnění tohoto požadavku využívá specializovaných obvodů. S narůstajícími požadavky na flexibilitu sítí rostou i požadavky na flexibilitu těchto obvodů. Provádět změny v jazycích pro popis hardwaru je však složité a časově náročné. Tato práce se proto zabývá implementací obvodů pro extrakci a následné spojení položek hlaviček paketů s využitím vysokoúrovňové syntézy.

Abstract

Almost every device on the network needs to extract some fields from the packet headers for its operation, perform operations on them, and forward the reassembled packet. This processing must be implemented at a speed corresponding to the line speed. On high-speed networks, specialized circuits are used to meet this requirement. As the demands on network flexibility increase, so do the demands on the flexibility of these circuits. However, making changes to the hardware description languages is complex and time consuming. This work therefore deals with the implementation of circuits for extraction and subsequent merging of packet header items using high-level synthesis.

Klíčová slova

HLS, FPGA, Extrakce, Spojování, Internetový provoz

Keywords

HLS, FPGA, Extraction, Merging, Internet traffic

Citace

BRÁZDA, Mikuláš. *Hardwarová akcelerace extrakce a spojování položek z hlaviček paketů.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Hardwarová akcelerace extrakce a spojování položek z hlaviček paketů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Tomáše Martínka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Mikuláš Brázda
10. května 2022

Poděkování

Nejprve bych rád poděkoval vedoucímu mé bakalářské práce za odbornou pomoc a cenné rady. Dále bych chtěl poděkovat členům týmu Liberouter, zejména pak Bc. Marku Bencovi, Ing. Jakubu Cabalovi a Ing. Martinu Špinlerovi

Obsah

1 Úvod **2**

2 Teoretický úvod **4**

2.1 Popis problematiky 4
2.2 Současný stav 6
2.3 Kritika současného stavu 8
2.4 Vysokoúrovňová syntéza 9
2.5 Popis dostupného hardwaru 12

3 Návrh a implementace **16**

3.1 Extrakce hlaviček 16
3.2 Extrakce payloadu 22
3.3 Editace či vkládání hlaviček 23
3.4 Vložení payloadu 24
3.5 Výsledky implementace 25

4 Testování **29**

4.1 Simulační testování 29
4.2 Testování v hardwaru 31

5 Závěr **33**

Literatura **34**

Kapitola 1

Úvod

Síťové modely rozdělují zařízení, služby i software počítačových sítí do skupiny vrstev dle architektury. Při odesílání dat po počítačové síti k nim musí každá vrstva přidat dodatečné informace, aby je bylo možné doručit. Může se jednat například o informace o odesílateli a/nebo o příjemci. Přídání takových dat se nazývá zapouzdření. Zapouzdřením dat vzniká datová jednotka shopna přenosu po síti, chcete-li paket.

Každé zařízení v síti při příjmu paketu analyzuje jeho obsah a na základě této analýzy se rozhodne, jak s ním naloží. Například monitorovací sonda může pomocí vyextrahovaných polí identifikovat tok a inkrementovat jeho čítače přenesených bajtů či paketů. V kontextu růstu rychlosti počítačových sítí se jeví využití klasických procesorů pro analýzu paketů jako nedostačující, zejména pak pro linky dosahující rychlosti 10 Gb/s a více. Z tohoto důvodu se ke zpracování paketů často využívá specializovaného hardwaru. Vývoj a verifikace takového hardwaru však trvá poměrně dlouho a proto je v něm složité reflektovat nové standardy.

V průběhu let za účelem zvýšit flexibilitu síťového hardwaru vznikly konfigurovatelné obvody, které jsou schopny extrahovat velké množství různě velkých hlaviček. Konfiguraci obvodů lze provádět například jazykem P4 (Programming Protocol-independent Packet Processors), který slouží k popisu síťových zařízení. Nevýhodu takových řešení představuje pevný počet těchto konfigurovatelných obvodů. Maximální počet hlaviček, které bude zařízení schopno zpracovávat, odpovídá počtu těchto obvodů. Další nevýhodou může být i fakt, že takový obvod kvůli své univerzálnosti, zabírá na čipu stejně velký prostor pro všechny typy hlaviček. Integrovaný obvod navržený pro konkrétní aplikaci se nazývá ASIC (Application Specific Integrated Circuit).

Nekteří vývojáří se proto zaměřili spíše na vývoj s využitím FPGA (Field Programmable Gate Array), které neslouží pouze pro konkrétní aplikace, ale lze je naprogramovat dle potřeb. Díky této technologii tak obvod zabírá na čipu jen potřebný prostor pro konkrétní aplikaci a nevyužitá část pak může provádět jinou činnost. Na základě popisu zpracovávaných protokolů například v jazyce P4 se vygeneruje popis hardware. Základem takových implementací jsou generické bloky kódu, jejichž parametry se nastaví dle požadavků uživatele. Každý blok extrahuje hlavičku jednoho protokolu a pro každdou vrstvu počítačových sítí se vygeneruje jedna úroveň zřetězení. Právě zřetězením lze značně urychlit, jinak sekvenční zpracování paketů. Správným nastavením zřetězení lze docílit požadované latence a/nebo frekvence zpracování. Avšak nalezení ideálního řešení není jednoduché a často vyžaduje využití nějaké heuristiky.

Jak již bylo zmíněno, vývoj hardwaru a jeho následná verifikace zabírají velké množství času, a proto se v různých oblastech začíná využívat vysokoúrovňová syntéza. Vysokoúrov-

ňová syntéza převádí program popsaný v jazyce vyšší úrovně do jazyků pro popis hardwaru. Nástroje provádějící tuto syntézu zajišťují automatické namapování požadavků uživatele na výsledný hardware. Uživatel takto může specifikovat frekvenci zpracování či již zmiňovanou úroveň zřetězení. Právě této vlastnosti chceme využít pro aplikaci vysokoúrovňové v oblasti počítačových sítí. Cílem práce je tedy navrhnout a naimplementovat obvody pro extrakci položek hlaviček paketů a jejich spojování s využitím vysokoúrovňové syntézy a následně je otestovat v dostupném hardwaru.

V kapitole 2 budou popsány klíčové části z nastudované literatury k této práci s rozborem a kritikou současných řešení. V další kapitole 3 se zaměříme na návrh řešení společně s jeho implementací. V poslední kapitole 4 bude popsáno testování komponent.

Kapitola 2

Teoretický úvod

Cílem této práce je vytvořit obvody pro extrakci a spojování položek z hlaviček paketů s využitím vysokoúrovňové syntézy a ověřit jejich funkčnost na dostupném hardwaru. V této kapitole bude nejprve uveden rozbor problematiky. Následně budou popsány existující řešení společně s jejich kritikou. Rovněž zde budou uvedeny základní principy vysokoúrovňové syntézy a popis hardwaru, ve kterém budou obvody testovány.

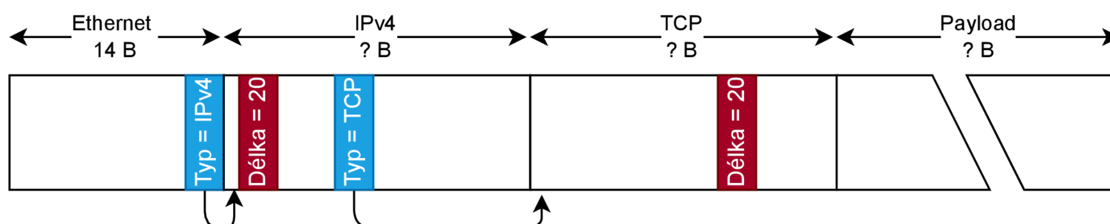
2.1 Popis problematiky

Protokol určuje sémantická a syntaktická pravidla pro komunikaci mezi nejméně dvěma uzly. Můžeme jej popsat formálně pomocí stavových automatů, gramatik, grafovými modely či algebraickými prostředky [16]. Model a architektura síťových transakcí mezi dvěma počítačovými systémy jsou popsány pomocí síťového zásobníku. V oblasti internetu se můžeme setkat se dvěma takovými modely: model OSI od organizace ISO a TCP/IP čili internetový model. Každý z těchto modelů rozděluje síťová zařízení, služby a software na skupiny vrstev dle architektury. Z tohoto důvodu se tyto modely označují jako vrstvové. Každá vrstva má své protokoly, které zajišťují její funkcionalitu. Při komunikaci odesílatel sestupuje síťovým zásobníkem shora dolů a každá vrstva zásobníku přidává *metadata*, doslova „data o datech“, která se označují jako hlavička protokolu. Tento proces se nazývá zapouzdření. Nejnižší vrstva poté přeposílá data spolu s metadaty příjemci. Na straně příjemce se prochází síťový zásobník zdola nahoru a jednotlivé vrstvy potřebná metadata naopak oddělují [21]. Pro pochopení složitosti extrakce byly nastudovány struktury hlaviček těchto protokolů: Ethernet [24], Multiprotocol Label Switching (MPLS) [10], virtual local area network (VLAN), Internet Protocol version 4 (IPv4), Internet Protocol version 6 (IPv6), Transmission Control Protocol TCP a User Datagram Protocol (UDP) [21]. Zde budou uvedeny hlavně stěžejní informace pro extrakci a spojování.

Každé zařízení v síti ke své činnosti potřebuje jedno nebo i více konkrétních polí z hlaviček paketu. Extrakce a identifikace těchto polí se anglicky nazývá parsing a komponenta, jež tuto operaci provádí se označuje parser. Délka a formát paketu se ovšem může libovolně měnit v závislosti na použitých technologiích, z čehož vyplývá, že pozice a dokonce i příslušnost dané hlavičky konkrétnímu paketu není předem známá. Jaká další hlavička je obsažena v příchozím paketu, se totiž dozvídáme až z aktuálně zpracované hlavičky např. pole Délka/Typ uvnitř ethernetové hlavičky. Ne všechny protokoly udržují informaci o následující hlavičce. Například protokol MPLS obsahuje pouze informaci o tom, zda se jedná o poslední MPLS štítek či nikoliv. V takových případech dochází k odhadování typu další

hlavičky na základě obsahu s využitím různých heuristik a nebo vyhledáním v tabulce dle příchozího portu [12].

Další problém při hardwarové akceleraci extrakce a spojování představují hlavičky s předem neznámou délkou. Velikost hlaviček protokolů IPv4 a TCP se pohybuje v rozmezí mezi dvaceti až šedesáti bajty. Takže kromě toho jaká hlavička bude následovat, předem nedokážeme odhadnout ani to, kde bude následovat. Na obrázku 2.1 můžeme vidět formát TCP paketu. Odesílatel zapouzdřil data TCP hlavičkou, IPv4 hlavičkou a Ethernet hlavičkou. Při extrakci však tuto informaci zjišťujeme postupně, což naznačují barevná pole.

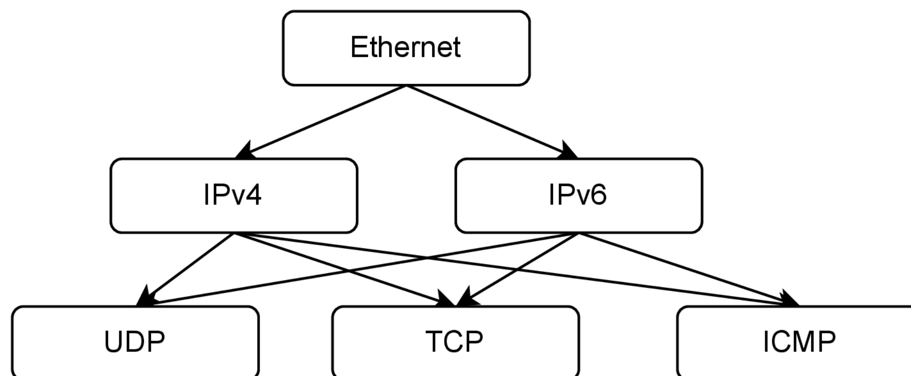


Obrázek 2.1: TCP paket sloužící k ilustraci zapouzdření hlaviček. Během extrakce začínáme od ethernetové hlavičky a postupně zjišťujeme pozici další hlavičky a její typ. Vlastní zpracování obrázku z [12].

Hlavička IPv6 [19] má sice pevně danou délku čtyřicet bajtů, ale za ní mohou následovat rozšiřující hlavičky IPv6, které situaci značně komplikují. Například rozšiřující hlavička **Volby pro všechny (Hop-by-hop options)** musí být zpracována každým zařízením, které implementuje IPv6. Tuto hlavičku můžeme interpretovat jako pole Volby u IPv4, avšak s tím rozdílem, že má maximální délku 65 535 B. Často tedy zařízení určuje vlastní maximální délku rozšiřujících hlaviček IPv6, kterou je schopno zpracovat. IPv6 kromě takto formátovaných rozšiřujících hlaviček nabízí i velké datagramy označované jako Jumbogramy. Jumbogramy můžou dosahovat délky 4 GB.

Velikost IP paketu je omezená linkou, která má vždy vlastní maximální přenosovou jednotku (Maximum transmission unit, MTU). Pro standard Ethernet tato jednotka odpovídá maximální velikosti datového pole uvnitř ethernetového rámce, což je 1500 bajtů. U větších IP paketů než je MTU se provádí fragmentace.

Extrakci hlaviček lze popsat jako průchod orientovaným acyklickým grafem (Directed Acyclic Graph, DAG) označovaným jako *Parse graf*. Graf zobrazuje sekvenční posloupnost extrakce. Uzly zde představují hlavičky protokolů a hrany sekvenční pořadí zpracování. Cílem komponenty pro extrakci je pak podporovat takový graf, který je schopen popsat co nejvíce různých kombinací hlaviček [12]. Konkrétní příklad takového grafu můžeme vidět na obrázku 2.2, který znázorňuje *Parse graf* se šesti extrahovanými hlavičkami.



Obrázek 2.2: Parse graf s hlavičkami Ethernet, IPv4, IPv6, TCP, UDP a ICMP

Po extrakci položek provede zařízení nad těmito daty nějaké operace a rozhodne se, jak má být s paketem naloženo. Může se jednat pouze o nějakou sondu, která dle vyextrahovaných dat identifikuje tok a inkrementuje jeho vnitřní čítače přenesených bajtů či paketů. S vyextrahovanými daty však pracují i mnohem komplexnější zařízení, jako jsou například směrovače, jež se dle IP adresy musí rozhodnout, kam daný paket poslat [12]. Rovněž musí například dekrementovat pole životnosti (Time To Live, TTL) uvnitř protokolů IPv4, IPv6 či MPLS a nakonec přepočítat kontrolní součty paketu a paket poslat dál.

Některá zařízení provádí pouze úpravu konkrétních bajtů vstupní sekvence dat. Jiná mohou požadovat větší změny v podobě přidání či odebrání některé hlavičky. Typickým příkladem může být práce přepínače s VLAN tagem na jednotlivých rozhraních. Takové operace musí posouvat vstupní data o různé množství bajtů. Tato skutečnost dělá ze spojování paketů náročnou operaci z pohledu spotřebovaných zdrojů, zejména pak v oblastech vysokorychlostních sítí. Proces editace či spojení upraveného rámce se anglicky označuje deparsing a komponenta, která tuto činnost zajišťuje, deparser.

Důležitou vlastnost pro návrh řešení představuje i rozšiřitelnost, jelikož nové standardy v oblastech počítačových sítí vznikají poměrně často [12].

2.2 Současný stav

Nové technologie s sebou přínášejí stále větší požadavky na zpracovávání paketů. Technologie jako jsou 4K přenos videa nebo internet věcí (IoT) požadují vysokou rychlost zpracování. Ovšem koncept softwarově definovaných sítí (Software Defined Networks, SDN), který narůstá stále více na popularitě, potřebuje vysokou flexibilitu a programovatelnost sítí [13].

Největší flexibilitu nabízejí softwarová řešení, avšak tato řešení dosahují rychlosti řádově 10 Gb/s [11] i při využití specializovaných instrukcí pro výpočty kontrolních součtů, bitových operací či vyhledávání [1].

Pro linky s rychlostmi řádově ve stovkách Gb/s musíme využít specializovaný hardware. Nabízí se tedy možnost řešení s využitím aplikačně specifického obvodu (ASIC) či programovatelného hradlového pole (FPGA). Jak již bylo zmíněno v úvodu práce, existují P4 konfigurovatelná zařízení [23] založená na univerzalních obvodech. Hlavní nevýhodou těchto univerzálních obvodů je režie, kterou způsobuje právě tato univerzálnost. Pro vyextrahování VLAN tagu který má 4 bajty musíme totiž použít stejný obvod jako pro vyextrahování IPv6 hlavičky s velikostí 40 bajtů. Z tohoto důvodu se zaměříme na řešení s využitím FPGA.

Hlavní dvě oblasti, které souvisejí s touto prací jsou proto zejména: hardwarová akcelerace zpracování paketů v FPGA s propustností alespoň 100 Gb/s a popis extrakce či spojování s využitím jazyků vyšší úrovně.

M. Attig a G. Brebner [2] ve své práci představili akceleraci extrakce s využitím FPGA, která dosahuje rychlosti až 400 Gb/s. Rovněž představili nový jazyk pro popis extrakce (Packet Parsing language, PP) se „syntaktickým cukrem“ jazyka Java pro deklaraci tříd. Z popisu v jazyce PP se vygeneruje pro každou úroveň síťového zásobníku jedna úroveň zřetězení. Protokoly, které se nacházejí na stejné úrovni budou zpracovávány paralelně. V každém stupni zřetězení tedy dojde k extrakci jedné hlavičky a v ustáleném stavu může být rozpracován stejný počet paketů jako je počet stupňů zřetězení. Parsery s datovou šířkou 1024 bitů spotřebovávají přes 10 % zdrojů dostupných v Xilinx Virtex-7 870HT FPGA a latence se pohybuje od 292 do 540 ns.

G. Brebner na základě znalostí o spojování paketů při rychlostech 10 Gb/s z [5] a extrakci při rychlostech 400 Gb/s z předchozí práce definuje nový jazyk PX pro popis celého zpracování paketů, tedy extrakci, aplikování akcí a následné spojení. Po vzniku jazyka P4 (Programming Protocol-independent Packet Processors) byl vytvořen kompilátor jazyka P4 do PX a vznikl nástroj Xilinx SDNet. Tento nástroj je schopen převádět P4 jazyk do VHDL a výsledné komponenty využívají průměrně dvojnásobný počet zdrojů oproti ručně napsaným implementacím.

V. Puš a další [18] upozorňují na problémy implementace v HDL jazycích, zejména na její časovou náročnost a proto ve své práci definují univerzální rozhraní parseru (Generic Protocol Parser Interface, GPPI) pro většinu protokolů. Tímto univerzálním rozhraním dosáhnou náznaku objektové nadstavby pro jazyk VHDL. Díky tomuto rozhraní se kód VHDL stává více udržitelným při změnách standardů či rozšiřitelným pro nové protokoly. V práci se také autoři zaměřují na zpracování více paketů v rámci jednoho vstupního slova, jelikož už při datové šířce 512 bitů a minimální velikosti ethernetového rámce 64 bajtů (512 bitů) k tomuto jevu může docházet. V případě kolize dvou stejných hlaviček v rámci jednoho datového slova dochází ke zpoždění o jeden hodinový takt. Stejně jako u předchozí práce se jedná o zřetězenou architekturu navíc s možností konfigurace, zda má být pro danou úroveň síťového zásobníku přidána nová úroveň zřetězení či nikoliv. Pro n vrstev nabízí tedy 2^n kombinací a současně s tím upozorňují na potřeby pečlivého prozkoumání co možná nejvíce možností, pro potřeby konkrétní aplikace. Jejich 400 Gb/s extraktor využívá pouze 4,88 % zdrojů dostupných v Xilinx Virtex-7 870HT FPGA.

P. Benáček a další [4] vytvořili generátor jazyka VHDL z vysokoúrovňového jazyka P4 (Programming Protocol-independent Packet Processors) s využitím GPPI z [18]. Pro každý protokol popsaný v jazyce P4 se generuje vlastní analyzátor, který obsahuje: blok pro samotnou extrakci využívající GPPI se správně nastavenými parametry, blok pro zjištění další hlavičky a také logiku pro výpočet posunu pro další analyzátor. Při generování byla zachována možnost specifikace konkrétního umístění registrů pro zřetězení. Bylo porovnáno 20 % všech možných konfigurací a spotřebované zdroje byly průměrně dvakrát vyšší u generovaných extraktorů než u těch ručně napsaných.

J. Cabal a další [7] navázali na předchozí dvě práce a generují z jazyka P4 parsery s propustnosti až 1 Tb/s. Autoři vytvořili novou sběrnici, která vkládá více rámců do vstupního slova a s využitím více paralelních extraktorů jsou schopni během jednoho hodinového cyklu tyto rámce zpracovat, díky čemuž nedochází k razantním poklesům propustnosti při malých rámcích.

J. S. Silva a další [20] vytvořili generátor extraktorů na základě popisu v jazyce P4 do C++, který následně pomocí nástroje Vivado HLS vysyntetizují do VHDL. Základ imple-

mentace tvoří C++ šablony, které umožňují tvoření generických tříd či funkcí. Extraktor je možné konfigurovat pro rychlosti od 10 Gb/s do 160 Gb/s. Obvod pracuje na frekvenci 312,5 MHz díky čemuž dosáhne propustnosti 100 Gb/s už při 320bitovém slově. Autor porovnává svoji implementaci i s překladačem jazyka P4 do VHDL z [4], který byl rozebírán v rámci jedné z předchozích kapitol. Zlepšení můžeme vidět ve spotřebě zdrojů i v latenci a to řádově v desítkách procent.

Firma Bittware [25], se rozhodla pro implementaci svého parseru s využitím vysokoúrovňové syntézy. V rámci tohoto článku se firma zabývá srovnáním implementace jejich parseru v jazyce P4 a v jazyce C++. Pro překlad jazyka P4 využívají Xilinx SDNet a pro syntézu C++ Xilinx Vivado HLS. Závěrem autorů je, že C++ nabízí lepší vyjadřovací schopnosti a více možností pro optimalizace, díky čemuž vzniká řešení s menším množstvím spotřebovaných zdrojů. Součástí článku bohužel není dostatečný popis řešení pro porovnání s ostatními implementacemi.

Předchozí práce popisují zejména extrakci paketů. Editace nebo složení celého rámce je možné provádět klasickými procesory, ale při vyšších rychlostech se vyplatí použít specializovaný obvod. P. Benáček a další rozšířili jejich předchozí práci zaměřenou na generování parserů [4] o generování deparserů. Architektura deparseru je podobná parseru, počet vrstev zřetězení opět odpovídá počtu podporovaných vrstev zásobníku. Každá vrstva může být taktéž vypnuta či zapnuta. Oproti parseru jsou bloky uspořádány od shora dolů, tedy nejvyšší vrstva síťového zásobníku je přidána jako první. Díky tomuto způsobu uspořádání si každá vrstva posune vstup pouze o velikost hlavičky a vkládá svá data vždy na pozici nula. Toto řešení pracuje pouze s jedním rámcem v jednom hodinovém taktu, takže při sdílených slovech dochází ke snížení propustnosti.

J. Cabal ve své diplomové práci [6] navrhl obvod, který využívá stejnou sběrnici jako 1 Tb/s extraktor [7]. Obvod využívá speciální jednotky Spacer, která zajišťuje přidávání či ubírání bajtů na sběrnici, tak aby mohly být prováděny operace jako je přidání či odebrání nějaké hlavičky. Rámce se tedy nesestavují znovu. Autor diskutuje možnost generování deparseru z popisu v jazyce P4, kterou poté s dalšími spoluautory [8] implementuje.

S. Ibanez a další [14] představili kompilátor jazyka P4 do otevřených NetFPGA karet. Využívají nástroje Xilinx SDNet.

T. Luinaud a další [15] představili otevřený generátor deparseru z jazyka P4 a do srovnání spotřebovaných zdrojů zahrnuli i práci z [4] a vygenerovaný deparser z nástroje Xilinx SDNet. Oproti [4] snížili počet spotřebovaných zdrojů pět krát a oproti deparseru z Xilinx SDNet dokonce desetkrát. Obě porovnávané implementace přitom dosahují nižší propustnosti.

2.3 Kritika současného stavu

Vývoj v HDL jazycích je časově náročný a pro popis vysokorychlostních obvodů je zapotřebí velké úsilí. S rostoucím zájmem v softwarově definovaných sítí se očekává, že nové síťové protokoly budou vznikat čím dál častěji [18]. V minulých letech proto vznikla řada generátorů HDL jazyků z přepisu v jazyce P4 s využitím zřetězeného zpracování. Problém ovšem nastává při výběru nejlepšího řešení. Některé implementace generují fixní počet zřetězených úrovní [5], jiné nabízí možnost specifikovat, zda se má či nemá daný stupeň povolovat [4]. Fixní počet zřetězených úrovní může vést k vysoké latenci a/nebo k nízké frekvenci zpracování. Řešení v podobě variabilního nastavení nabízí velké množství možností a jejich otestování zabírá mnoho času.

Vysokoúrovňová syntéza nabízí automatické zřetězení dle požadovaných parametrů na frekvenci zpracování. Navíc jazyk C++ nabízí větší vyjadřovací schopnosti oproti jazyku P4 a tudíž lze vytvářet flexibilnější řešení. Tato práce tedy zejména zkoumá možnosti vysokoúrovňové syntézy v oblasti hardwarové akcelerace parseru a deparseru na vysokorychlostních sítích.

2.4 Vysokoúrovňová syntéza

Od počátku HDL jazyků se ustálil klasický způsob vývoje hardwaru. V těchto jazycích se vytvoří popis RTL schématu a převod do hardwaru se přenechává na automatizovaných nástrojích poskytující logickou syntézu. V oblasti elektroniky však zaznamenáváme neustálý vývoj. Dle Moorova zákona se počet tranzistorů na čipu každý rok zdvojnásobí. S narůstajícím počtem tranzistorů roste i množství funkcí, které může čip poskytovat. Tuto funkcionalitu však musí někdo naprogramovat a proto vzniká tlak na zvýšení produktivity vývojářů [9].

Zvýšení produktivity může být docíleno poskytnutím vyšší úrovně abstrakce v podobě jazyků vyšší úrovně. Mnoho firem proto v průběhu let vytváří automatizované nástroje pro převod mezi algoritmickým popisem a RTL schématem [17]. Proces převodu mezi těmito úrovněmi abstrakce se nazývá výsokoúrovňová syntéza (High Level Synthesis, HLS) a většinou se skládá z těchto pěti operací [9]:

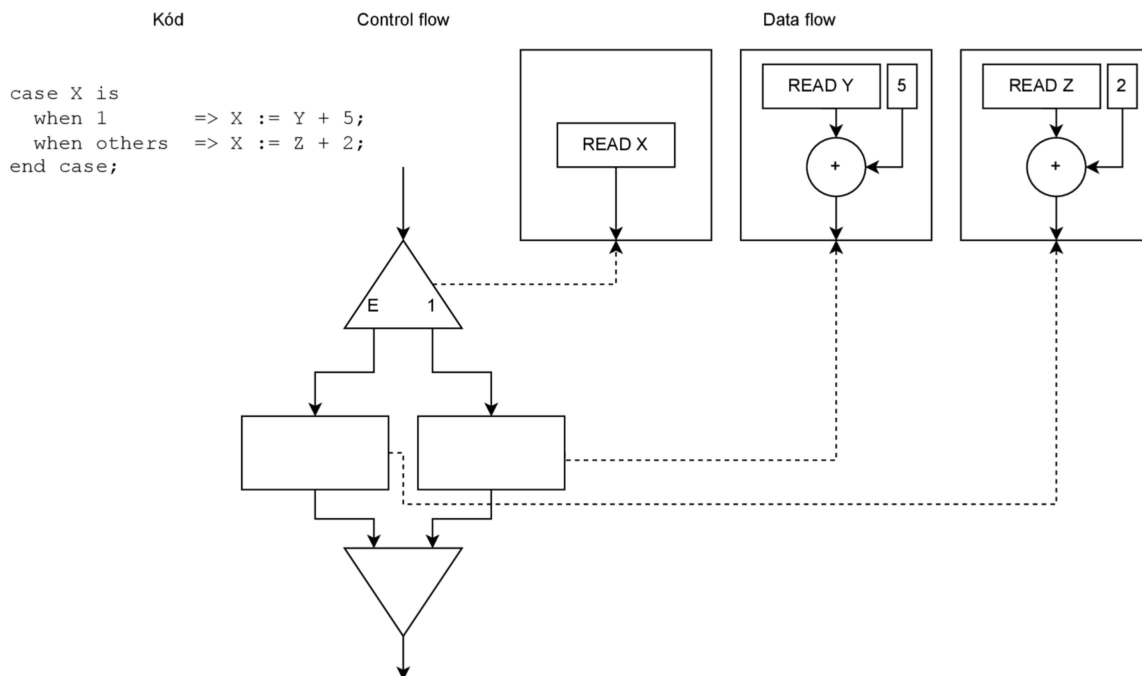
1. **Kompilace a modelování** – Na začátku každý HLS nástroj provede přeložení kódu do formální reprezentace, která snadno dokáže reprezentovat závislosti mezi operacemi. Formální reprezentaci zde představuje Control Data Flow Graph (CDFG). Uzly zde tvoří konkrétní operace a hrany mezi nimi tvoří datové závislosti. Větvení programu konstrukcemi *if* či *switch* je zde reprezentováno volitelnými hranami. Nástroj v tomto kroku provádí i optimalizace, mezi které patří výpočet konstantních hodnot, rozbalení smyček a eliminace mrtvého kódu či falešných datových závislostí. Obrázek 2.3 znázorňuje Control Data Flow Graph pro jednoduchý VHDL kód.

2. **Alokace prostředků** – Během alokace prostředků se určují typy potřebných prostředků a jejich počet pro splnění požadavků. Využívá se knihovna komponent, která musí poskytovat jejich důležité charakteristiky, jako je například zpoždění.

3. **Plánování operací** – V této části dochází k naplanování operací do hodinových cyklů. Operace může být rozvržena do jednoho či více hodinových cyklů v závislosti na počtu dostupných zdrojů a datových závislostech.

4. **Přiřazení prostředků** – Operace máme již naplánované a tudíž je můžeme přiřadit konkrétním hardwarovým komponentám.

5. **Generování RTL** – Po předchozích krocích již víme, které komponenty budou potřeba. Stačí tedy vygenerovat jejich RTL popis v HDL jazycích. Jedná se o funkční bloky kódu (jako jsou ALU, sčítačky, násobičky a další), paměťové či propojovací bloky (například sběrnice, třístavový budič, multiplexory).



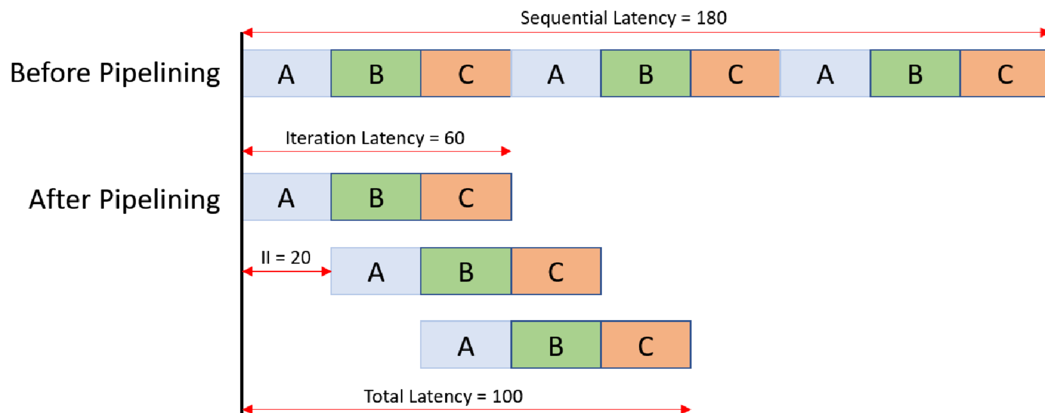
Obrázek 2.3: Znázornění Control Flow a Data Flow pro jednoduchý VHDL kód.

Většina firem poskytující nástroje pro logickou syntézu se rozhodla pro HLS framework založený na jazycích C, C++ či System C [3]. Mezi tyto firmy patří i společnost Xilinx, jejíž FPGA čip je součástí karty, na které se budou obvody implementované v této práci testovat. Nástroj se nazývá Vitis HLS a déle popsané principy se budou vztahovat k tomuto konkrétnímu nástroji, nicméně by se neměly značně lišit napříč nástroji konkurečních firem.

Zřetězené zpracování

Zřetězené zpracování (Pipelining) [28] je technika aplikovatelná v mnoha oblastech každodenního života. Jedná se o způsob jakým urychlit zpracování většího množství produktů.

Uveďmě si příklad zpracování rámce o hlavičkách A, B a C, jejichž zpracování trvá 20, 10 a 30 minut. Jedním extraktorem by sekvenční zpracování trvalo 60 minut. Kdybychom však využili tři zřetězené extraktory, první rámec by trval 60 minut, ale každý další, už pouze 30 minut. Latence zpracování by tedy byla 60 minut a inicializační interval 30 minut. Nutno podotknout, že inicializační interval se odvíjí od nejpomalejšího prvku. Právě z tohoto důvodu je vhodné rozdělovat kód na bloky o stejném inicializačním intervalu čímž dojde k nejlepšímu možnému zřetězení. Kdybychom byli schopni v předchozím příkladu rozdělit akce A, B, C tak, aby každá trvala celkem 20 minut, celková latence by byla stále 60 minut, ale inicializační interval by odpovídal 20 minutám. Zřetězené zpracování tří rámců by pak zabralo 100 minut namísto 180 minut u sekvenčního zpracování. Tento příklad můžeme vidět na obrázku 2.4.



Obrázek 2.4: Schéma zpracování tří rámců před zřetězením a po zřetězení. Převzato z [28].

Vitis HLS nabízí řadu direktiv s prefixem #PRAGMA HLS. Direktivy slouží pro optimalizaci programu. Mezi tyto direktivy patří i #PRAGMA HLS PIPILENE II=*hodnota*. Tuto direktivu stačí použít v hlavní funkci implementace. Na pozici *hodnota* se pak doplní očekávaný inicializační interval v hodinových taktech a nástroj se postará o zřetězení. Plné propustnosti dosáhne implementace při II=1, čehož se v rámci této práce budeme snažit dosáhnout.

Aby bylo možné splnit požadavky na zřetězení nesmí vznikat datová závislost mezi iteracemi. Taková datová závislost vzniká při současném přístupu do paměti více iteracemi, kde alespoň jeden přístup je zápis. Programátorovi je umožněno vzniklé datové závislosti označit za falešné pomocí direktivy #PRAGMA HLS DEPENDENCE.

Dostupné knihovny

Vitis HLS nabízí ke svému nástroji C++ knihovny [29], které usnadňují vývoj. V této práci jsou používány dvě z těchto knihoven a to *HLS stream* a *Arbitrary precision data types*.

První ze jmenovaných knihoven obsahuje šablonovou třídu *hls::stream<>*, která poskytuje funkce pro práci s rozhraním typu FIFO, handshake nebo AXI. Komunikaci mezi konzumentem a producentem lze provádět blokujícími i neblokujícími metodami. Blokující čtení způsobí zastavení konzumenta dokud se neobjeví data ve frontě. Blokující zápis zastavuje producenta dokud se fronta neuvolní. Tyto metody však musí pozastavit celé zřetězené zpracování a vedou na problémy s časováním. Proto se v této práci používají pouze neblokující metody s pomocnými mechanismy pro kontrolu hloubky front.

Druhá z knihoven nabízí šablonové třídy *hls::ap_uint<>* a *hls::ap_int<>*. Obě tyto třídy poskytují rozhraní pro práci s datovými typy, jejichž velikost je zarovnána na jednotky bitů. *hls::ap_uint<>* slouží pro neznaménkový obor hodnot a *hls::ap_int<>* pro hodnoty se znaménkem.

Verifikace kódu

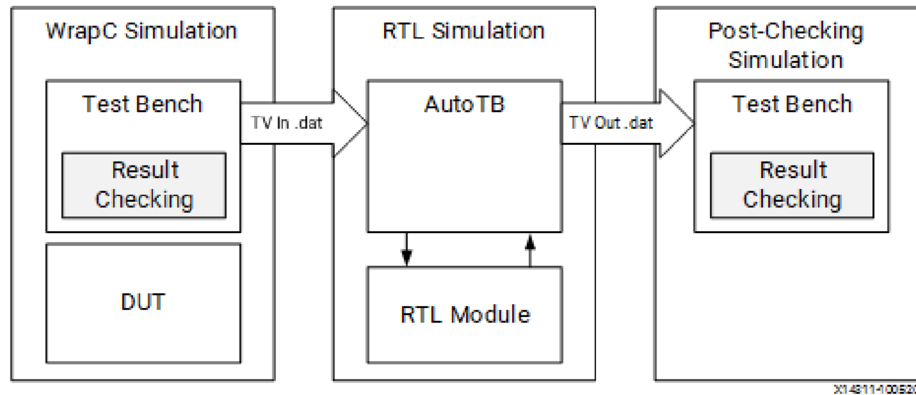
Nespornou výhodou vysokoúrovňové syntézy je i testování. Simulaci, stejně jako implementaci, můžeme popsat v jazyce C++. Tento kód není syntetizován a tudíž na rozdíl od popisu hardwaru nemá žádná omezení. Vitis HLS nabízí pro verifikaci kódu dvě různé simulace:

1. C simulace – ověření funkčnosti kódu v jazyce C před syntézou do RTL schématu,

2. C/RTL co-simulace – ověření správné funkčnosti vygenerovaného RTL schématu.

Program pro testování implementace se nazývá *test bench*. Základ takového programu tvoří funkce *main()*, ve které se volá hlavní funkce implementace. *test bench* by měla při úspěchu navracet nulu a jinak nenulové číslo.

C simulace jednoduše přeloží *test bench* do strojového kódu dostupným překladačem a výsledný program spustí. V takové simulaci lze ladit program pomocnými výpisy či využít ladící režim umožňující krokování programu. Po úspěšné C simulaci můžeme vysyntetizovat kód a provést C/RTL co-simulaci, jejíž kroky jsou vidět na obrázku 2.5.



Obrázek 2.5: Schéma C/RTL co-simulace. Převzato z [26].

Obrázek se skládá z těchto kroků [26]:

1. Spuštění C simulace (WrapC Simulation) nad implementací v C++ (DUT), při kterém se ukládají vstupní transakce (TV In.dat);

2. Vstupní transakce vstupují do obálky pro RTL simulaci (AutoTB) vygenerované nástrojem VivadoSimulator nebo jiným nástrojem třetí strany. Obálka je napojena na RTL schéma ve VHDL či Verilogu (RTL Module);

3. Výsledky RTL simulace (TV Out.dat) jsou zpřístupněny funkci *main()*, která je porovnává s výsledky C simulace.

Stejně jako při normální simulaci HDL jazyků lze zobrazit signály testované komponenty.

2.5 Popis dostupného hardwaru

Tým Liberouter [27], pracující pod záštitou sdružení CESNET, nabíží vývojovou platformu pro zpracování síťového provozu (Network Development Kit, NDK), která umožňuje uživateli snadno a rychle vyvíjet nová síťová zařízení založené na akceleračních kartách s čipem FPGA. Platforma umožňuje škálování na 10, 100 i 400 Gigabitový Ethernet a její součástí je software i firmware. Hlavními moduly firmwaru jsou:

- **Ovladač DDR4 pamětí** umožňuje přístup uživatelské aplikaci k externím pamětem.

- **Síťový modul** ze zachycených ethernetových paketů vyextrahuje pouze rámec a zkontroluje jeho kontrolní součet.

- **Blok uživatelská aplikace** představuje libovolnou uživatelem naprogramovanou aplikaci.

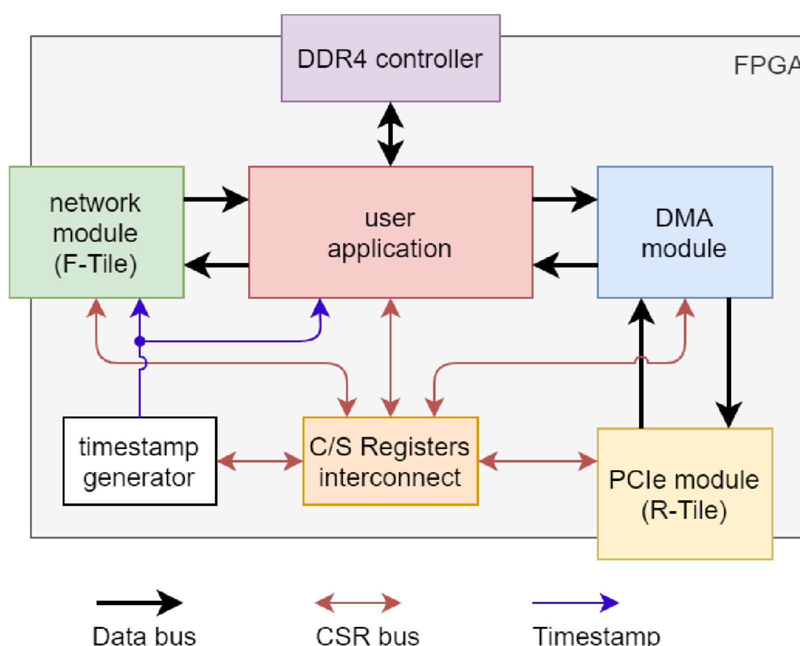
- **Modul pro přímy přístup do paměti (Direct Memory Access, DMA)** poskytuje vysokorychlostní přenos přes PCI-Express rozhraní (až do PCIe Gen5 x16). Při využití PCIe Gen 5 x16 přenáší data rychlostí 400 Gb/s. Jelikož software nedokáže zpracovávat ani generovat takové rychlosti najednou, využívá k tomu více jáder procesoru. Každé jádro tedy generuje provoz do vlastního DMA kanálu.

- **Generátor časových značek** zajišťuje synchronizovaný čas uživatelské aplikaci a síťovému modulu.

- **Modul přistupující k registrům komponent** umožňuje jednotný přístup k registrům většiny komponent ze softwaru s využitím přímého přístupu k PCIe zařízení (FPGA kartě). Požadavky přijaté z modulu PCIe rozděluje do jednotlivých komponent.

- **Modul PCIe** představuje sadu komponent pro řízení všech požadavků na PCIe.

Aby byla zachována propustnost komponent i při velice krátkých rámcích, moduly mezi sebou komunikují pomocí sběrnice podporující více rámců uvnitř datového slova (Multi Frame Bus, MFB). Schéma zapojení komponet lze vidět na obrázku 2.6.



Obrázek 2.6: Moduly firmwaru z vývojové platformy týmu Liberouter pracujícího pod záštitou sdružení CESNET. Převzato z [22].

Jak již bylo zmíněno platforma neobsahuje pouze firmware, ale i software. Součástí softwaru jsou i tyto nástroje příkazového řádku:

- *nfb-boot* – nahrává vysyntetizovaný firmware do karty,

- *nfb-info* – zobrazuje základní informace o kartě včetně aktuálně nahraného firmwaru,

- *nfb-dma* – dotazuje se na stav konktrétních DMA kanálů,
- *nfb-eth* – nastavuje ethernetové transceivery a dotazuje se na jejich stav či na počet přenesených rámců,
- *nfb-bus* – zasílá MI (Memory Interface) požadavky,
- *ndp-tool* – umožňuje zasílat a přijímat pakety ve formátu PCAP. Mimo jiné lze tyto operace provádět i pomocí konkrétních příkazů *ndp-read, ndp-receive, ndp-transmit* a dalších.

Sběrnice pro přenos více rámců

Přenášená data přes tuto sběrnici jsou organizována do jednotek nazývaných slova. Slovo může být rozděleno na několik regionů (Region) obsahující určitý počet bloků (Block). Bloky se dělí na položky (Item). Konkrétní konfigurace je možné vytvářet pomocí parametrů v tabulce 2.1. Pro pozice konců a začátků rámců platí tato pravidla:

1. Region smí obsahovat pouze jeden začátek a jeden konec rámce.
2. Začátek rámce musí být zarovnán na začátek bloku.
3. Konec rámce musí být zarovnán na položku.

Název	Popis
REGIONS	počet regionů uvnitř slova
REGION_SIZE	počet bloků uvnitř každého regionu
BLOCK_SIZE	počet položek uvnitř každého bloku
ITEM_WIDTH	šířka každé položky v bitech

Tabulka 2.1: Generické parametry MFB

Velikost regionu $REGIONS*REGION_SIZE*BLOCK_SIZE*ITEM_WIDTH$ by měla odpovídat minimální velikosti rámce. Zkráceně se konkrétní konfigurace popisuje jako MFB#(REGIONS, REGION_SIZE, BLOCK_SIZE, ITEM_SIZE).

Konkrétní konfiguraci lze potom odvodit dle definovaných standardů. V této práci budeme zapojovat obvody na 100GbE linku, která definuje tyto pravidla pro ethernetové rámce:

1. jsou zarovnány na bajty,
2. musí začínat na násobcích 8 bajtů,
3. jejich minimální velikost je 64 bajtů.

Z čehož vyplývá, pro obvody v této práci, konfigurace MFB#(X,8,8,8). X můžeme regulovat podle frekvence obvodu a cílené propustnosti. Jelikož většina obvodů v projektu Liberouter dosahuje frekvencí 200 MHz a chceme dosáhnout teoretické propustnosti 100 Gb/s stačí nám zvolit X=1. Šířka vstupu tedy odpovídá 512 bitům a požadovaná frekvence komponent bude 200 MHz. Komponenty budou komunikovat se sběrnicí pomocí signálů uvedených v tabulce 2.2.

Název	Směr	Popis
DATA	vysílač → přijímač	přenášená data
SOF_POS	vysílač → přijímač	pozice začátku rámce pro jednotlivé regiony
EOF_POS	vysílač → přijímač	pozice konce rámce pro jednotlivé regiony
SOF	vysílač → přijímač	příznak začátku rámce pro jednotivé regiony
EOF	vysílač → přijímač	příznak konce rámce pro jednotivé regiony
SRC_RDY	vysílač → přijímač	příznak připravenosti odesílatele k odesílání dat
DST_RDY	přijímač → vysílač	příznak připravenosti příjemce k příjmu dat

Tabulka 2.2: Signály MFB

Transakce mezi přijímačem a odesílatelem začíná, když jsou oba signály SRC_RDY i DST_RDY nastaveny na hodnotu jedna. Při deaktivaci signálu DST_RDY, odesílatel drží hodnotu na sběrnici. V případě, že nemá odesílatel data k odeslání (SRC_RDY = 0), pak je na sběrnici nedefinovaná hodnota. Sběrnice garantuje generování rámců délších než je šedesát čtyři bajtů a správnou posloupnost signálů SOP a EOP.

Aby bylo možné rozšířit bloky pro extrakci a spojení implementované v rámci této práce o akce, součástí rozhraní jsou i metadata v podobě délky rámce, časové známky, vstupního a výstupního portu (identifikující DMA kanál nebo ethernetový port), které při pouhé extrakci a spojení zůstávají nezměněny. Tato metadata jsou platná vždy při začátku rámce. Součástí rozhraní jsou také MI (Memory Interface) požadavky, které zprostředkovávají komunikaci s modulem pro přístup k registrům.

Kapitola 3

Návrh a implementace

Každá konkrétní aplikace může požadovat zpracování různých protokolů a dokonce i různá pole hlaviček těchto protokolů. V takovém případě by bylo nutné pro nové požadavky napsat i nové funkce pro extrakci. Úprava kódu v jazyce C/C++ je sice značně jednodušší oproti HDL jazykům, ale i tak může být potenciálním zdrojem chyb. Navíc pro každý nový protokol by bylo vždy nutné vytvářet další kód a řešení by se časem stalo neudržitelné. Abychom snížili množství požadovaného kódu pro konkrétní aplikaci vyextrahujeme vždy celou hlavičku a pak konkrétní aplikace zvolí, jaká data jsou z ní potřeba. S takovým přístupem lze hlavičky odlišovat pouze podle velikosti. Problémem z hlediska návrhu obvodů jsou pak hlavičky s variabilní délkou. Nabízí se extrahovat vždy maximální délku a podle reálné délky ukončit zpracovávání dříve. V implementaci však takový způsob zpracování prozatím není podporován.

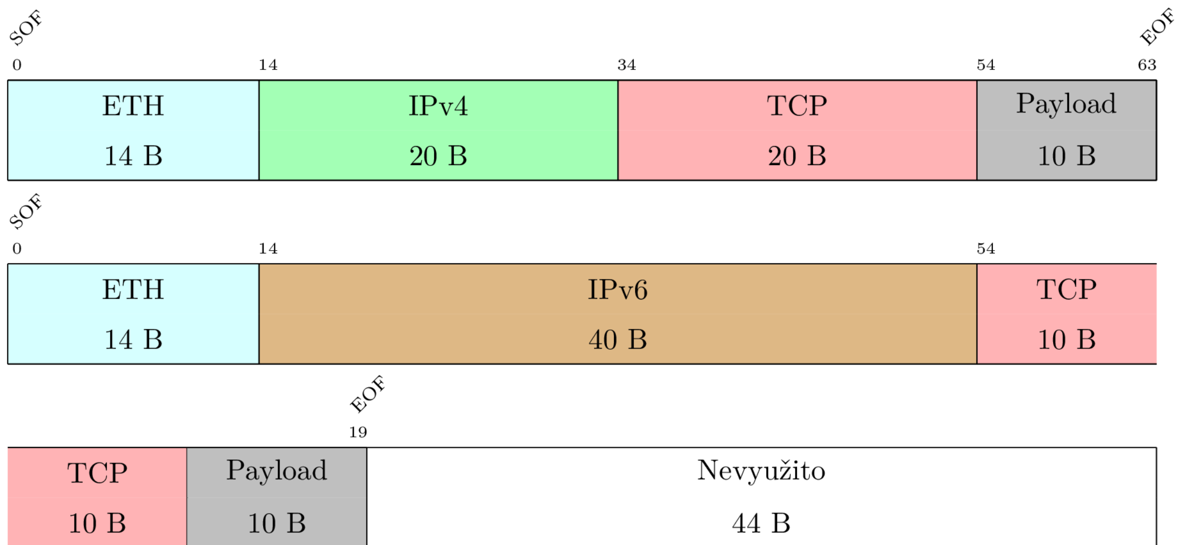
Výsledkem implementace jsou šablonové funkce *hlavičkový extraktor* pro extrakci hlavičky a *hlavičkový editor* pro vložení hlavičky, které umožní zpracování všech protokolů, jejichž hlavička je fixní délky.

Abychom byli schopni znovu sestavit celý rámec i v případě, že daná aplikace požaduje změnu některých polí, musíme si uchovávat i data, která nejsou pro tuto aplikaci potřebná. Nepotřebnými daty rozumíme payload, který následuje za posledním vyextrahovaným protokolem. Pro aplikaci, která extrahuje pouze Ethernet a IPv4, bude extrahován IPv4 payload pro ethernetové rámce typu IPv4 a Ethernet payload pro ostatní typy rámců (například IPv6). Pro účely extrakce i spojení těchto dat proto vznikly taktéž dvě komponenty *payload extraktor* a *payload editor*.

V této kapitole bude nejprve popsána návrh komponent pro extrakci polí z rámců a poté komponent pro sestavování rámců. V poslední sekci pak bude popsána výsledná implementace a její způsob použití.

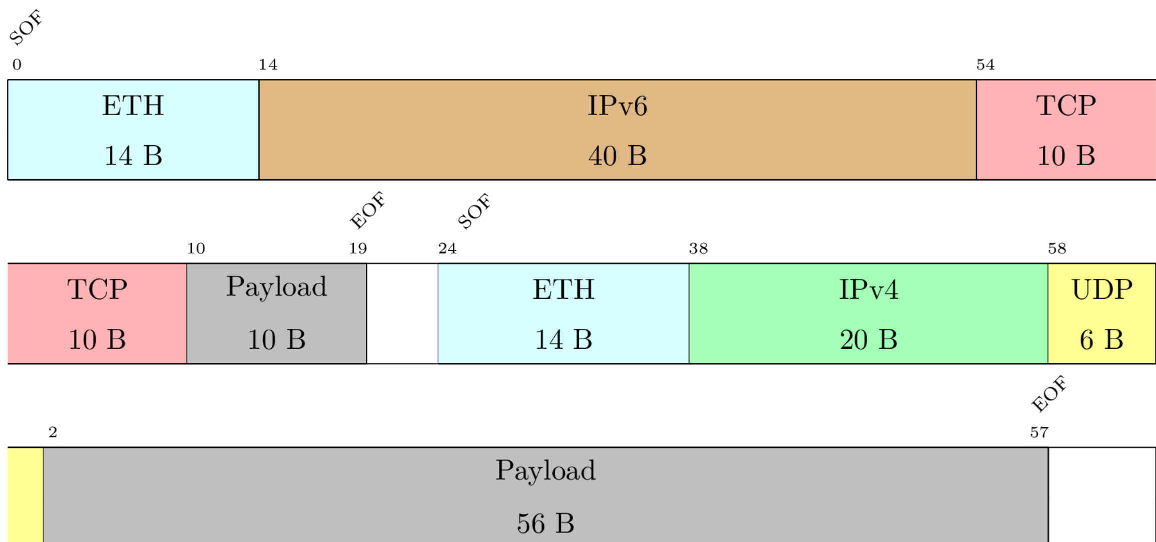
3.1 Extrakce hlaviček

Abychom mohli vyextrahovat hlavičku musíme získat její pozici. Komponenta pracuje na ethernetové síti takže na začátku rámce bude vždy hlavička Ethernet. Pozici hlavičky Ethernet získáme tedy ze vstupní sběrnice jako pozici začátku rámce. Další hlavička se bude nacházet hned za ethernetovou hlavičkou. Na obrázku 3.1 můžeme vidět vstupní MFB transakce obsahující dva rámce různé délky.



Obrázek 3.1: Tři transakce vstupní sběrnice obsahující dva rámce.

Oba rámce začínají na pozici nula. První z nich má délku šedesát čtyři bajtů a je přesně zarovnán uvnitř vstupního slova. Druhý z rámců svojí délkou osmdesát čtyři bajtů přesahuje délku vstupního slova a zabírá tak více než jedno slovo. Lze si například všimnout, že hlavička protokolu TCP uvnitř druhého rámce přesahuje přes dvě slova. Pakliže by rámce směly začínat pouze na pozici nula, mohli bychom tuto situaci označit, za nejhorší možnou. Při konfiguraci MFB#(1,8,8,8) však rámce můžou začínat na libovolné pozici, která je násobkem osmi bajtů. Z tohoto důvodu může uvnitř slova jeden rámec končit a druhý začínat. Takový případ lze vidět na obrázku 3.2.



Obrázek 3.2: Dva rámce sdílející jedno vstupní slovo

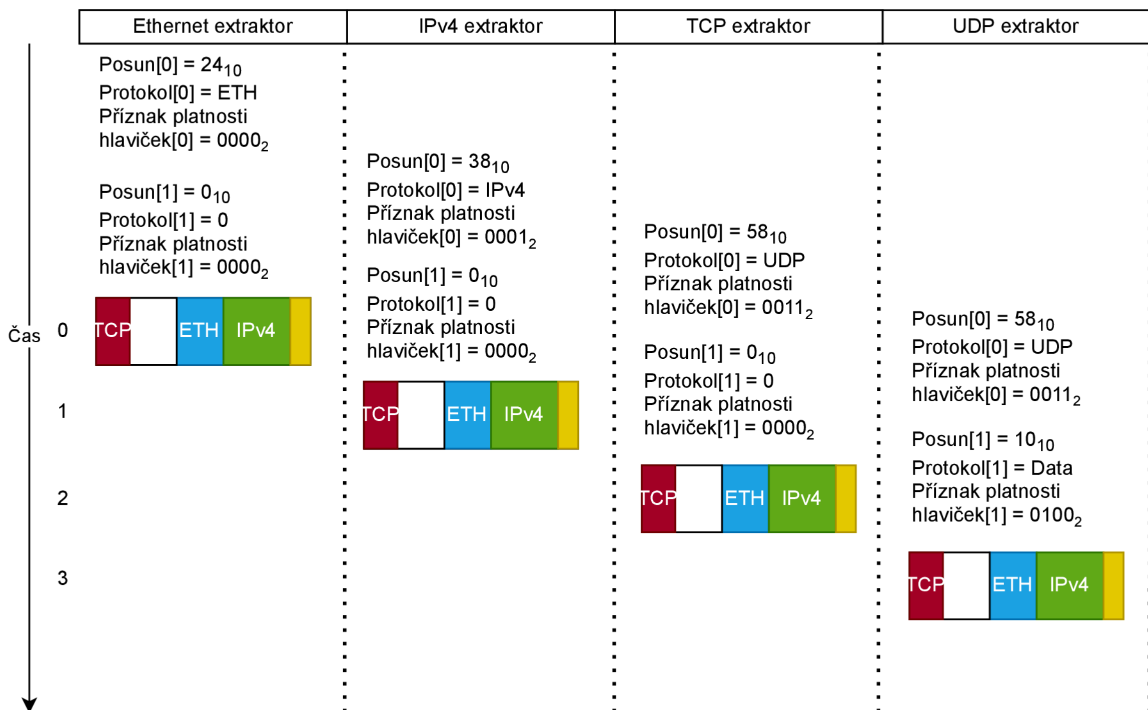
Metadata vstupního slova

Pozice je tedy jednou z informací, kterou si musí jednotlivé extraktory předávat mezi sebou. Né každý paket bude obsahovat celou sadu podporovaných protokolů, proto k pozici začátku hlavičky musí extraktory přidat ještě *identifikátor protokolu*, který se má zpracovávat a *vektor s příznaky platnosti hlaviček* s délkou odpovídající celkovému počtu extrahovaných protokolů. Identifikátor protokolu vyčtou extraktory z vlastní hlavičky. U hlavičky protokolu Ethernet to je pole **Délka/Typ** u IPv6 pole **Další hlavička** atd. Typicky však aplikace nemusí podporovat všechny možné protokoly. Právě proto se extraktoru předává pole obsahující identifikátory všech protokolů, jež se mohou za daným extraktorem vyskytovat. Ve vektoru příznaků platnosti hlaviček náleží každému extraktoru jedna pozice, odpovídající jejich pořadí v rámci celého zpracovávání, jejíž hodnotu nastavuje na jedna po vyextrahování hlavičky.

Jelikož se na reálných sítích může stát, že některé zařízení násilně ukončí rámec uvnitř některé z hlaviček, další informací přenášenou mezi extraktory je *příznak chyby*.

Slovo může obsahovat data až dvou rámců, tedy musí tyto metadata udržovat dvakrát. První slouží pro nově začínající rámec a druhé pro končící rámec.

V případě, že není sběrnice připravená odesílat data (SRC_RDY = 0), musí být slovo označeno jako neplatné. Tuto informaci si extraktory předávají mezi sebou v podobě *příznaku platnosti slova*, který je poslední informací uvnitř metadat slova. Na obrázku 3.3 můžeme vidět naznačení zpracování druhého vstupního slova z obrázku 3.2 čtyřmi extraktory. Pro jednoduchost obrázku budeme brát v potaz, že sběrnice má po celou dobu SRC_RDY = 1 a jedná se o platný rámec, tedy zanedbáme *příznak platnosti slova* a *příznak chyby*. V čase nula extraktor Ethernetu zjistí, zda se nenachází identifikátor Ethernetu na pozici jedna, tedy zda končící rámec neobsahuje hlavičku Ethernet. Poté se podívá na pozici nula, což představuje nově začínající rámec. Zde vidíme, že se nachází hodnota identifikující Ethernet. Takže Ethernet extraktor vyextrahuje svoji hlavičku (14 bajtů) z pozice začátku rámce (24) a nastaví hodnotu nového posunu na $24 + 14 = 38$. Rovněž musí nastavit identifikátor dalšího protokolu na IPv4 a příznak platnosti Ethernetu na 1. V čase jedna zjistí extraktor IPv4 podle identifikátoru protokolu, že se uvnitř daného slova nachází nový rámec s IPv4 hlavičkou. Stějně jako extraktor Ethernetu vyextrahuje svoji hlavičku (20 bajtů) a nastaví nový posun $38 + 20 = 58$, příznak hlavičky IPv4 na 1 a identifikátor protokolu na UDP. V čase dva obdrží tato data TCP extraktor. TCP extraktor si však uchovává z předchozího zpracovávání vnitřní informaci, že došlo k přetečení jeho hlavičky. Musí tedy nejprve vyextrahovat zbytek své hlavičky a aby nepřepsal metadata na pozici nula musí nastavit příznak platnosti hlavičky, další protokol a posun v metadatech na pozici jedna. Během své aktivity také musí zkontrolovat metadata na pozici nula, jelikož by se zde mohla opět nacházet jeho hlavička. Identifikátor protokolu je však UDP, tudíž ukončuje svoji činnost a předává data dál. V čase tři pak UDP extraktor vidí na pozici jedna identifikátor payloadu a na pozici nula identifikátor svého protokolu. Z tohoto důvodu se pokusí vyextrahovat svoji hlavičku (8 bajtů), která však přetekla do dalšího slova ($58 + 8 = 66$) a tudíž si uvnitř musí uložit vnitřní informaci, že došlo k přetečení jeho hlavičky.



Obrázek 3.3: Ukázka zpracování zarovnaného rámce čtyřmi extraktory

Samotná extrakce hlavičky

Jelikož se rámec může skládat z různých hlaviček a také může začínat na různých pozicích uvnitř slova, měli bychom předpokládat, že se hlavička může nacházet na libovolné pozici uvnitř vstupního slova.

Na extrakci budeme tedy potřebovat komponentu, která zvládne vstupní datový blok posunout o libovolný počet bajtů. V hardwaru se pro tuto operaci běžně používá válcový posouvač. Válcový posouvač je obvod složený z multiplexorů s N datovými vstupy, N výstupy a $log_2(N)$ řídícími vstupy, kde N odpovídá šířce rotovaného bloku v bitech. Řídící vstupy určují rotace datových vstupů.

Při programování softwaru by zřejmě nebylo nijak složité takovou komponentu popsat. Potřebujeme totiž překopírovat data o velikosti hlavičky z obecně velkého pole na pozici *offset*. Taková operace lze provést pomocí smyčky s pevným počtem opakování. Aby však syntetizátor mohl odhadnout počet iterací smyčky a mohl ji rozbalit, musí být počet iterací znám už při syntéze. Tedy řešení by mohlo vypadat například takto:

```
1  template<int WORD_SIZE, int HDR_SIZE>
2  void barrel_shifter(char src[], char dst[], unsigned char offset)
3  {
4      for(int i=0; i<WORD_SIZE; i++)
5      {
6          if (i > offset and i < offset + HDR_SIZE)
7              dst[i-offset] = src[i];
8      }
9  }
```

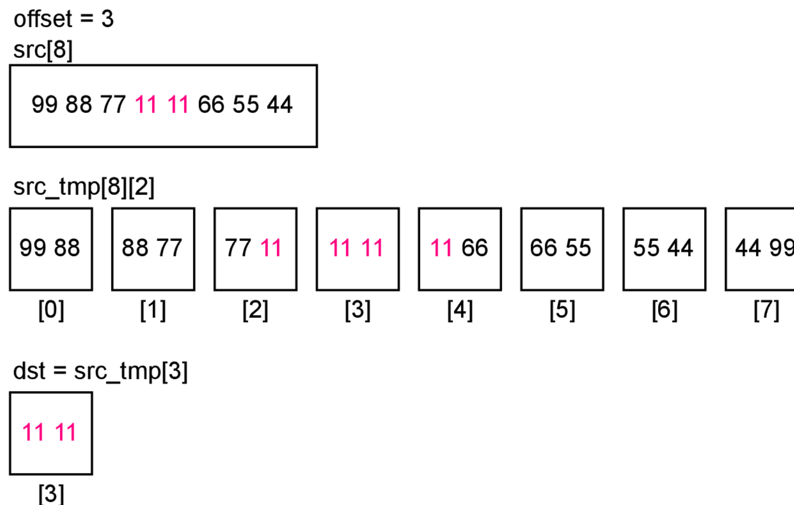
Výpis 3.1: Implementace válcového posouvače v softwaru

Ukázalo se, že takové řešení požaduje velké množství zdrojů. Hlavním důvodem těchto nároků je předem neznámá hodnota proměnné *offset*, která zapříčiní vznik sčítačky/odečítačky pro každý bajt. Při velikosti slova šedesát čtyři bajtů vzniká tedy šedesát čtyři sčítaček. Abychom snížili počet spotřebovaných zdrojů na tyto sčítačky, vytvoříme implementaci ilustrovanou ve výpisu kódu 3.2. Na řádku čtyři vidíme dvojrozměrné pole se stejným počtem řádků jako je velikosti slova (*WORD_SIZE*). Na řádek i se překopíruje slovo rotované o i bajtů doprava (řádek 6–11). Na výstup pak zapíšeme pouze data ze slova, které leží na pozici *offset* (řádek 12–14).

```
1  template<int WORD_SIZE, int HDR_SIZE>
2  void barrel_shifter(char src[], char dst[], unsigned char offset)
3  {
4      ap_uint<8> src_tmp[WORD_SIZE][HDR_SIZE];
5
6      for (int i = 0; i < WORD_SIZE; i++) {
7          for (int j = 0; j < HDR_SIZE; j++) {
8              unsigned char idx = (j+i) % WORD_SIZE;
9              src_tmp[i][j] = src[idx];
10         }
11     }
12     for (int i = 0; i < HDR_SIZE; i++) {
13         dst[i] = src_tmp[offset][i];
14     }
15 }
```

Výpis 3.2: Řešení válcového posouvače v hardwaru

Na obrázku 3.4 vidíme, jak bude vypadat extrakce hlavičky o velikosti dva bajty ze vstupního pole o velikosti osm bajtů. Hlavičku zde představuje sekvence dvou po sobě jdoucích hexadecimálních číslic 11.



Obrázek 3.4: Extrakce dvoubajtové hlavičky ze vstupního slova o velikosti osm bajtů a s posunem o tři bajty. *src_tmp* je dvourozměrného pole obsahující na každém řádku hlavičku extrahovanou z pozice, která odpovídá indexu řádku.

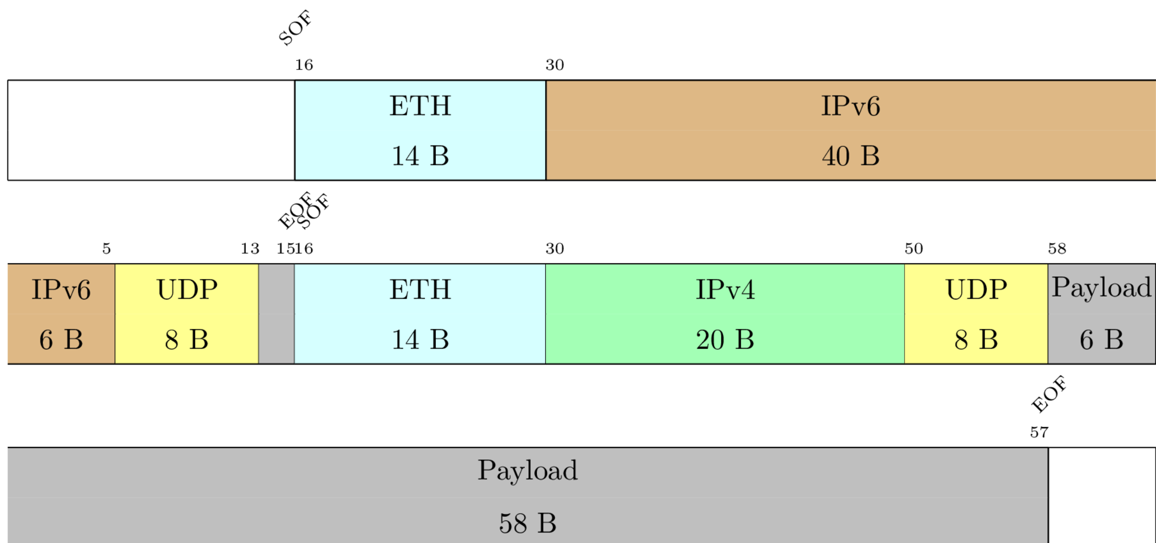
S pomocí válcového posouvače jsme tedy schopni rotovat vstupní slovo o libovolné množsví bitů, čímž dojde k zarovnání hlavičky na pozici nula. Díky tomu, že hlavičky protokolů jsou zarovnány na bajty můžou mít multiplexory vždy osmkrát méně vstupů i výstupů.

Dvě stejné hlavičky uvnitř vstupního slova

To, že hlavička protokolů může přesahovat přes okraj slova bylo naznačeno již v předchozích odstavcích. Tento problém by bylo možné vyřešit uchováním rozpracované hlavičky uvnitř extraktoru. Dva rámce uvnitř jednoho slova s sebou, ale přináší ještě jeden problém. Při velmi malých rámcích totiž může dojít k situaci, kdy vstupní slovo obsahuje dvě stejné hlavičky viz obrázek 3.5. V takovém případě bychom museli být schopni dokončit rozpracovanou hlavičku a ještě ve stejném cyklu stihnout i rozpracovat druhou.

Každá z hlaviček však leží na jiném posunu, takže bychom potřebovali dva válcové posouvače. Toto řešení však spotřebovává dvojnásobné množství zdrojů. Když však vezmeme v potaz, že vstupní rámec musí mít minimální délku šedesát čtyři bajtů, což je velikost vstupního slova, lze dojít k závěru, že v celkovém průměru bude vycházet jedna hlavička na jedno vstupní slovo. Maximálně se totiž může vyskytovat právě jeden rámec na jedno vstupní slovo a každý rámec smí obsahovat pouze jednu hlavičku konkrétního extraktoru.

Při kolizi dvou stejných hlaviček může tedy extraktor nejprve zpracovat první a až v dalším taktu tu druhou, což znamená, že extraktoru musí být zpřístupněna data z přechozího zpracování. Abychom při takové kolizi nemuseli pozastavovat zpracování, musí s jedním slovem v danou chvíli pracovat dva extraktory. Vstupní slovo slouží extraktorům pouze ke čtení a tudíž zde nedochází k datovým závislostem. Co extraktory však modifikují jsou metadata. Pro obrázek 3.5 musí v jednom běhu UDP extraktor nastavit další protokol, posun atd. pro končící rámec a ve druhém běhu pro nově začínající rámec. Ovšem linka se nepozastavuje, takže UDP extraktoru přišel opět další vstup, který musí zpracovat. Pro tyto účely byla přidána třetí sada metadat ke slovu. Tato sada tedy předává uvnitř aktuálních metadat aktualizované informace o předchozím slově. Konkrétně informuje o nově začínajícím rámci.



Obrázek 3.5: Dva nezarovnané rámce tak, že druhé vstupní slovo obsahuje dvě UDP hlavičky.

Na obrázku 3.3 si můžeme všimnout, že v případě, kdy pracuje IPv4 extraktor, tak Ethernet extraktor už je připraven číst další vstup a díky tomuto vzniká prostor pro zřetězení. Právě z důvodu zřetězení musí být na výstupu extraktorů fronta vyextrahovaných hlaviček.

3.2 Extrakce payloadu

Extraktor payloadu je komponenta, která následuje za posledním extraktorem hlavičky. Jeho úkolem je označit pozici/pozice payloadu v rámci vstupního slova. Reaguje na speciální hodnotu identifikátoru protokolu uvnitř metadat. Tuto hodnotu nastavují extraktory v případě, že jejich pole uvnitř hlavičky, identifikující další protokol, neobsahuje identifikátor žádného z podporovaných protokolů a nebo při chybě během extrakce. Do této komponenty vstupují tedy data z posledního extraktoru. Aby byla schopna správně označit pozici/pozice payloadu i v případě dvou stejných hlaviček uvnitř slova, musí s těmito daty vždy pracovat až o takt později. Díky tomuto zpoždění bude komponenta schopna přečíst informace o předchozím slovu i z aktuálních metadat, kam se mohou dostat právě v případě kolize dvou stejných hlaviček uvnitř jednoho slova.

Payload může zabírat více slov a proto si musí extraktor udržovat vnitřní stav zpracování. Jestliže přišel identifikátor začátku payloadu a nepřišel konec rámce, pak extraktor přechází do stavu, kdy všechna další slova označuje jako payload. V případě, že se na vstupu objeví konec rámce musí být ověřeno, zda se uvnitř slova nenachází začátek payloadu dalšího rámce. Pakliže ano, komponenta setrvává v tomto stavu, jinak přechází do stavu, kdy čeká na začátek payloadu. Z každého rámce tak vzniká fronta payloadu.

Součástí výstupního slova jsou i doplňující informace. Stejně jako u hlaviček i začátek payloadu se může nacházet na libovolné pozici výstupního slova a tudíž nesmí tato informace chybět. Součástí slova může být i začátek druhého payloadu a z toho důvodu obsahuje slovo

i příznak, zda se tam nachází či nikoliv. Payload končí s koncem rámce. Protože konec rámce smí být ve slově jen jeden, můžeme předpokládat, že tomu tak bude i s koncem payloadu.

Jak již bylo zmíněno, komponenta se nachází za posledním hlavičkovým extraktorem a tudíž má kompletní informace o vstupním rámci. Právě z tohoto důvodu má na starosti ještě jednu funkci a to předání vektoru s příznaky platnosti hlaviček. V příchozích metadatech můžou být informace až o dvou rámcích, komponenta zajišťuje předání nejprve informaci o končícím rámci a poté až o začínajícím. Stejně jako hlavičky a užitečný obsah je i tato informace ukládána do výstupní fronty.

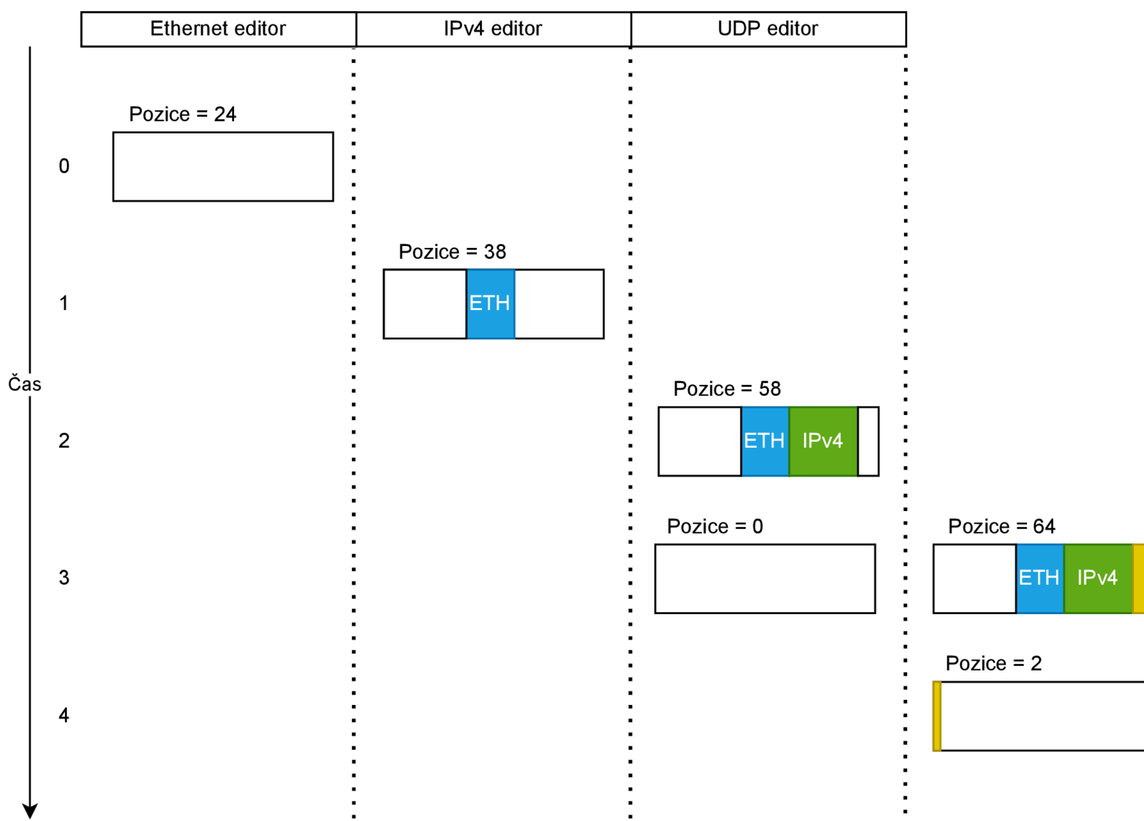
3.3 Editace či vkládání hlaviček

Po extrakci dat následují akce, které mohou upravit rámec. Jednoduché akce mohou vyžadovat jen přepsání hodnoty na konkrétní pozici vstupního slova. U složitějších akcí si však s pouhou editací nevystačíme a často je výhodnější složit celý rámec znovu. Příkladem složité operace, která již byla uvedena v popisu problematiky, může být vložení či odebrání VLAN tagu. Při takové operaci musí být posunuta všechna data za Ethernet hlavičkou. Abychom byli schopni sestavit rámec bez ohledu na to, jaká akce sestavení předcházela, budeme ho vždy sestavovat znovu. Komponentu pro připojení jedné hlavičky budeme nazývat hlavíčkový editor.

Jak bylo uvedeno při popisu extrakce, začátek hlavičky se může nacházet na libovolné pozici a tudíž každý editor musí tuto vlastnost podporovat. Pro účely vkládání se tak opět využívá válcového posouvače. Abychom mohli vkládání provádět, musíme vědět pozici první hlavičky. Z tohoto důvodu se uchovává vstupní značka začátku rámce. První editor, Ethernet editor, vkládá tedy hlavičku od pozice původního začátku rámce. Některé operace můžou způsobit, že se předchozí rámec prodloužil nebo zkrátil a tudíž by pozice začátku dalšího rámce měla tuto změnu reflektovat. Z tohoto důvodu je možné pozici začátku posunout dle posunu konce předchozího rámce. Tuto informaci nám poskytne blok provádějící akce spolu s hlavičkami.

Editor Ethernetu po dokončení vkládání aktualizuje pozici začátku další hlavičky o velikost své hlavičky a spolu s upraveným slovem ji předá dál. Takto slovo putuje přes všechny editory. Během vkládání může dojít k zaplnění slova a to v případě, že hlavička některého z protokolů přesahuje přes dvě slova. Jelikož se na vstupu může v dalším taktu objevit nový rámec, musí být editor schopen vytvořit nové slovo, aby do něj uložil zbytek hlavičky. V tomto stavu má editor dvě slova, které musí zapsat na výstup. Proto v současném taktu zapíše editor zaplněné slovo a v dalším taktu zapíše slovo nově vzniklé. K této situaci však může docházet opakovaně a proto byly editory navrženy s frontou požadavků na vstupu. Editory pak jednoduše při přetečení hlavičky jeden takt nečtou ze vstupní fronty.

Do editoru vstupují slova z předchozího editoru a taky jeho hlavičky. Ke každému rámci jsou vygenerované všechny hlavičky a podle příznaku platnosti hlaviček, který se vytváří během extrakce a je možné jej modifikovat během akcí, se nastavuje jejich platnost. Z pohledu editoru to vypadá tak, že na vstupu má vždy hlavičku, která nemusí být platná. V případě, že není platná, editor posílá vstupní slovo na výstup. V opačném případě vkládá svoji hlavičku do vstupního slova. Obrázek 3.6 znázorňuje sestavení druhého rámce z obrázku 3.2. Můžeme zde vidět, že hlavička UDP přesahuje přes okraj výstupního slova. Editor UDP proto v čase tři vytváří nové slovo a zapíše do něj zbytek hlavičky, který se nevešel do předchozího slova.



Obrázek 3.6: Ukázka spojení hlaviček druhého rámce z obrázku 3.2. UDP extraktor v čase tři vytváří nové slovo z důvodu přetečení jeho hlavičky.

Jednou z akcí, provedenou před spojením rámce, může být i zahození rámce. Jelikož jsou akce prováděny pouze nad vyextrahovanými hlavičkami, musí dojít k zahození až při spojování rámce. Při akci bychom totiž nebyli schopni zahodit i užitečný obsah. Na začátku spojování dojde k označení rámce příznakem zahození. Mimo jiné rámec může být označen k zahození i v případě chyby při extrakci. K chybě totiž může dojít v libovolném extraktoru a zahozeny musí být i všechny již vyextrahované hlavičky.

3.4 Vložení payloadu

Příznak zahození zpracovává komponenta umístěná za posledním editorem nazývaná *Payload deparser*. Jejím úkolem je připojení payloadu rámce ke spojeným hlavičkám a případné zahození rámce. Payload není zpracováván uvnitř akcí a tudíž vstupuje do této komponenty v podobě, v jaké byl vyextrahován. Komponenta má tedy na vstupu frontu payloadu z extraktoru a frontu výstupů posledního editoru. Tyto fronty jsou skládány do MFB transakcí.

Stejně jako při extrakci payloadu, i zde je zapotřebí uchovávat vnitřní stav zpracování. Nejprve se přečte slovo z posledního editoru. Pakliže je uvnitř slova místo, přechází se do stavu, kdy se zpracovává payload. Během zpracovávání payloadu dochází ke kopírování dat ze vstupní fronty do rozpracovaného slova. Zaplněné slovo se vždy zapíše na výstup. Po přijetí konce payloadu, neboli konce rámce, se ověřuje zda posun uvnitř rozpracovaného slova není větší než padesát pět. Rámce totiž můžou začínat pouze na násobcích osmi a tudíž by se sem nový rámec už nevešel a slovo by v takovém případě bylo zapsáno na výstup.

Následně se přechází opět do stavu kdy čteme výstupní slovo posledního z extraktorů. Díky tomu, že rámce začínají na původní pozici, můžeme nově přečtená data opět připojit do rozpracovaného slova. Pro případ kdy akce nemění velikost rámce se tedy díky zachovávání původních pozic datového obsahu a začátku rámce obejdeme bez válcového posouvače. Válcový posouvač je zde však umístěn a je na uživateli, zda jej pomocí šablonového parametru zapne či nikoliv. Díky této možnosti můžeme pro jednoduché aplikace snížit počet spotřebovaných zdrojů.

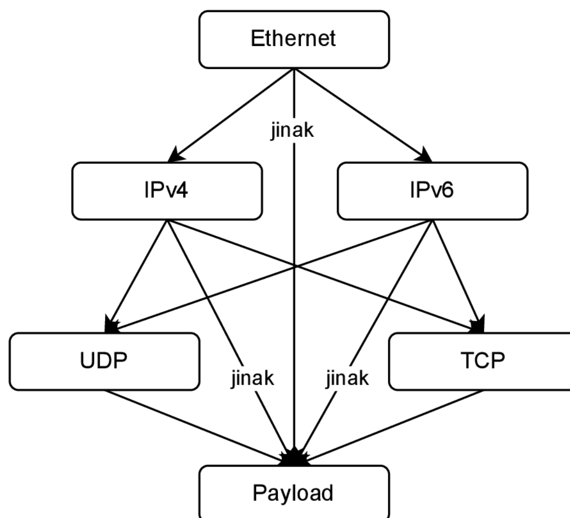
3.5 Výsledky implementace

Pro konkrétní aplikace vznikly konfigurovatelné komponenty v podobě šablonových funkcí. Jejich princip činnosti byl popsán v předchozích sekcích.

Extrakce

Pro konkrétní aplikaci stačí vytvořit funkci, nazveme ji *parser()*, která správně poskládá šablonové funkce a přidá k nim logiku pro sjednocení výstupních front. Správný způsob použití bude popsán v této sekci.

Jak již bylo popsáno v popisu problematiky extrakce představuje průchod *Parse grafem*. Konkrétní aplikace má svůj vlastní graf, kterým se musíme řídit při skládání komponent. Popis konstrukce konkrétní aplikace bude popsán na konkrétním grafu zobrazeném na obrázku 3.7. Do grafu byly přidány i hrany (s názvem *Jinak*), které představují cestu pro jiné identifikátory protokolů než ty, které chceme extrahovat. Tyto hrany směřují vždy do uzlu Payload, který představuje extrakci payloadu. Díky těmto hranám graf přijímá všechny druhy paketů.



Obrázek 3.7: Parse graf použitý pro vysvětlení způsobu použití komponent. Hrany *jinak* představují přechody pro všechny ostatní protokoly.

Pro každý uzel grafu představující hlavičku některého z protokolů budeme potřebovat šablonovou funkci se jménem *header_extractor()*. Komponenty musí být poskládány za sebe v pořadí v jakém jsou zobrazeny v Parse grafu. Pakliže leží dva uzly na stejné úrovni (pro

obrázek 3.7 například uzly IPv4 a IPv6), můžeme libovolně určit jejich vzájemné pořadí. Šablonové parametry funkce *header_extractor()* jsou popsány v tabulce 3.1.

Název	Popis
HDR_WIDTH_BYTES	velikost hlavičky v bajtech
PROTOCOL	identifikátor protokolu, který má být zpracováván
NO_HDRS	celkový počet extrahovaných hlaviček
ORDER_IN_THE_PIPELINE	pořadí v rámci všech hlavičkových extraktorů
NEXT_PROTO_POS	pozice pole, které určuje další protokol uvnitř hlavičky
NEXT_PROTO_WIDTH_BYTES	šířka pole, které určuje další protokol
NO_NEXT_PROTOS	počet podporovaných protokolů, které se můžou nacházet za touto hlavičkou
LAST_HDR	příznak, že za danou hlavičkou už nemůže být žádný z podporovaných protokolů

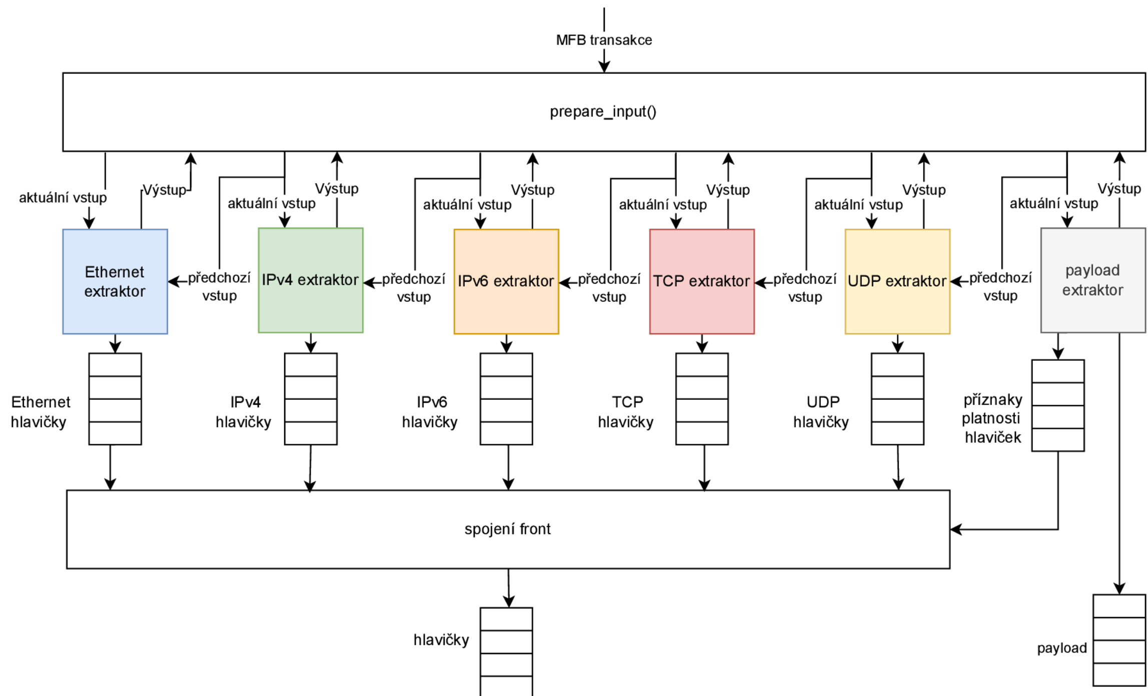
Tabulka 3.1: Šablonové parametry komponenty pro extrakci hlaviček

Celkem budeme pro náš příklad potřebovat pět funkcí *header_extractor()*. Pro každý extraktor musíme vytvořit pole, které bude obsahovat identifikátory protokolů, jejichž hlavičky se můžou za daným extraktorem nacházet. Z Parse grafu tuto informaci představují hrany bez označení *Jinak*. Pro náš Parse graf v obrázku 3.7 to bude pro protokol Ethernet pole se dvěma prvky IPv4 a IPv6. Pro protokoly IPv4 a IPv6 to budou pole se dvěma prvky UDP a TCP.

Kromě extrakce hlaviček musíme zajistit i extrakci payloadu. K tomuto účelu slouží šablonová funkce *payload_parser()*, která se umístí vždy za poslední funkci *header_extractor()*. Funkce přijímá pouze jeden šablonový parametr **NO_HDRS** určující celkový počet extrahovaných hlaviček.

Pro každý blok uvnitř linky budeme potřebovat registr. Registr lze snadno v jazyce C++ popsat pomocí statické proměnné. Abychom byli schopni zpřístupňovat všem blokům předchozí vstup musíme vytvořit vstupní pole o velikosti odpovídající celkovému počtu uzlů v Parse grafu (šest) plus jedna. Celkem tedy budeme mít pole *in_words[7]* a *in_metadata[7]*. Každý blok pak může měnit metadata takže musí mít i svůj výstup *out_metadata[6]*. Tato pole budou vytvořena uvnitř funkce *parser()*. Každému extraktoru se pak předává jako parametr prvek z pozice odpovídající jeho pořadí uvnitř linky v podobě aktuálního vstupu a prvek z pozice plus jedna v podobě předchozího vstupu. Nějak však musí být zajištěno aby se při každém volání funkce *parser()* přečetlo ze vstupu a data uvnitř vstupních polí se posunula vždy o jednu pozici. O tuto funkcionalitu se stará funkce *prepare_input()* na začátku linky. Její šablonový parametr je opět **NO_HDRS**.

Na obrázku 3.8 můžeme vidět všechny bloky extrakce pro Parse graf z obrázku 3.7 zapojené dohromady. Blok pro spojení front musí být specifický pro každou aplikaci, protože pro něj nelze najít univerzální rozhraní.



Obrázek 3.8: Schéma zapojení extrakčních bloků pro Parse graf z obrázku 3.7

Funkce *parser()* byla vysyntetizována ve dvou konfiguracích: **simple** – extrahuje hlavičky protokolů ethernet a IPv4, **full** – extrahuje hlavičky protokolů Ethernet, 2x VLAN, IPv4, IPv6, TCP a UDP. Pomocí nástroje Vitis HLS 2021.1 bylo vygenerováno RTL schéma této funkce s inicializačním intervalem jedna a nastavenou periodou 5 ns. Toto schéma bylo následně vysyntetizováno nástrojem Vivado 2021.1. Výsledky syntézy jsou uvedeny v tabulce 3.2. Zdroje, které funkce nespotřebovává nejsou uvedeny.

	LUT	FF	latence	frekvence [MHz]
simple	5 880	5 543	4	276
full	17 300	11 104	4	201

Tabulka 3.2: Výsledky syntézy funkce *parser()*.

Spojování

Opět byla definována šablonová funkce pro hlavičky fixní délky se jménem *header_editor()*. Funkce má pouze dva šablonové parametry: **PROTOCOL** - identifikátor dané funkce sloužící k rozlišení jedlnotlivých instancí šablon překladačem a **HDR_WIDTH_BYTES** - udávající velikost hlavičky.

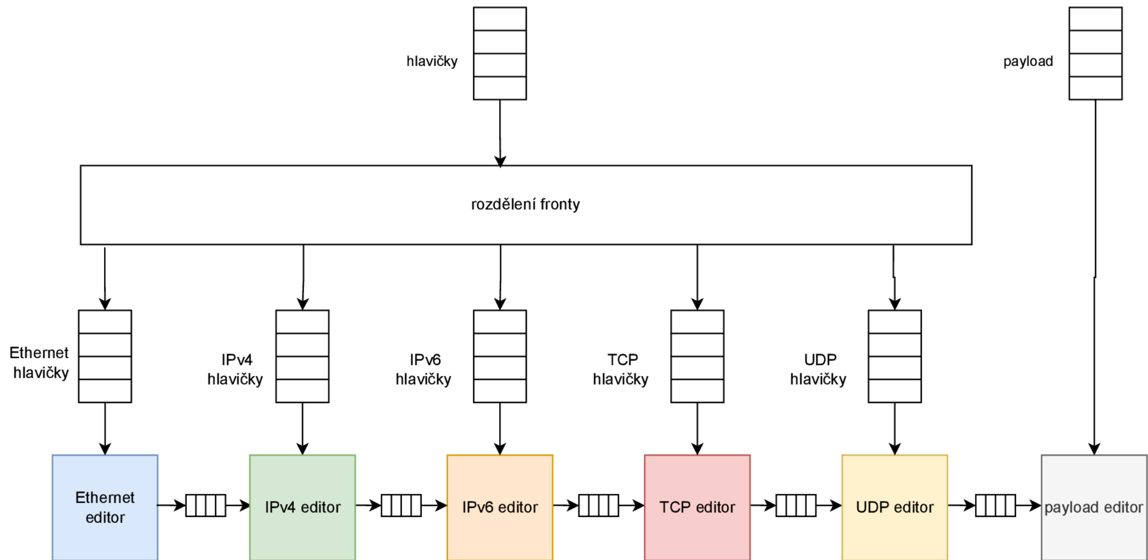
Pro spojování si nadefinujeme opět funkci, řekněme *deparser()*, do které naskládáme funkce *header_editor()* stejně jako u funkce *parser()*.

Oproti extrakci zde může docházet ke zpoždění zpracování a tudíž každý editor má na svém rozhraní tři statické fronty. Jedna představuje vstupní frontu hlaviček, druhá vstupní frontu slov a třetí výstupní frontu slov. Fronty jsou implementovány třídou *hls::stream*. Výstupní fronta slov vždy představuje vstupní frontu slov pro následující editor. Jelikož

dostaneme všechny hlavičky pohromadě, musí i funkce *deparser()* obsahovat logiku, specifickou pro každou aplikaci, která zajistí rozdělení vstupních hlaviček do jednotlivých front.

Za posledním editorem pak opět následuje zpracování payloadu tentokrát s využitím funkce *payload_deparser()*. Jediný parametr zde představuje příznak **NEW_OFFSET** určující, zda se má použít válcový posouvač, protože se provádějí akce, které mohou změnit rámce či nikoliv.

Na obrázku 3.9 můžeme vidět všechny bloky spojování pro Parse graf z obrázku 3.7 zapojené dohromady. Blok pro spojení front musí být specifický pro každou aplikaci, protože pro něj nelze najít univerzální rozhraní.



Obrázek 3.9: Schéma zapojení spojovacích bloků pro Parse graf z obrázku 3.7

Stejně jako u extrakce byla provedena syntéza funkce *deparser()* s výsledky v tabulce 3.3 a to v totožných konfiguracích.

	LUT	FF	BRAM	latence	frekvence [MHz]
simple	5 337	4 850	35	7	258
full	13 108	13 743	35	17	282

Tabulka 3.3: Výsledky syntézy funkce *deparser()*

Kapitola 4

Testování

Testování implementace bude rozděleno na dvě části. Nejprve se kapitola zaměřuje na popis testování uvnitř simulace a poté v samotném hardwaru.

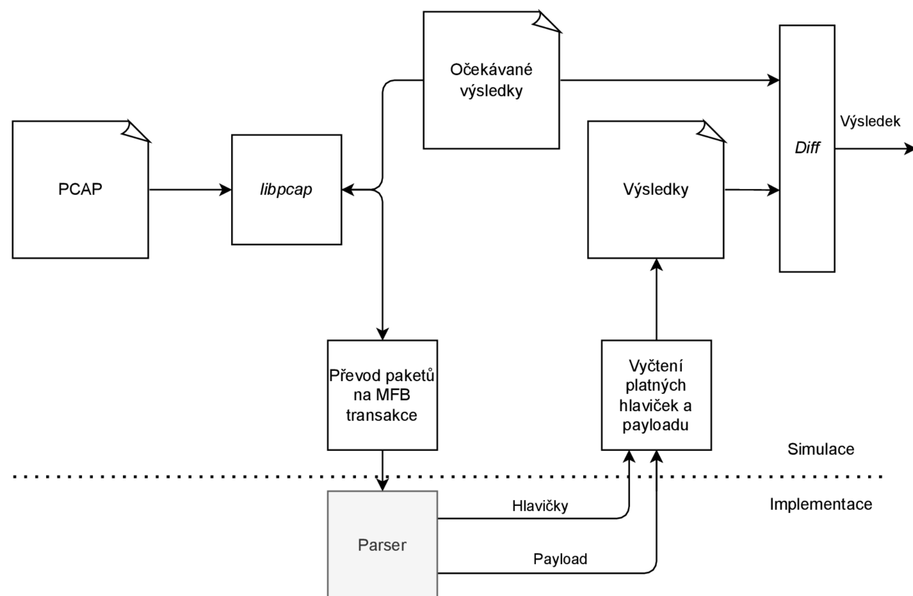
4.1 Simulační testování

V *test bench* extrakce potřebujeme vytvořit rámec a převést ho do posloupnosti MFB transakcí. Tedy potřebujeme vstupní rámce rozdělit do 64bajtových datových bloků se značkami začátku a konce rámce. Ruční vytváření většího množství vstupních transakcí by bylo však časově náročné a proto vzniklo automatizované prostředí pro jejich generování založené na knihovně *libpcap*. Jedná se o aplikační rozhraní pro záchyt síťové komunikace. Kromě záchytu živé komunikace umožňuje rovněž čtení ze souboru typu PCAP. V tomto formátu se ukládá síťová komunikace od linkové vrstvy výše. Právě soubor ve formátu PCAP je vstupem testovacích programů.

Knihovna *libpcap* navrací zachycené rámce, jako pole bajtů. Toto pole bajtů pak stačí rozdělit na 64bajtové bloky se značkami začátku a konce rámce. Pakliže značka konce rámce ukazuje na pozici menší než padesát šest, což je poslední možná pozice pro začátek dalšího rámce, vyplní se zbylá data novým rámcem. Takto vzniká v simulaci vstup extraktoru. Rámce jsou rovněž zapisovány do textového souboru, který představuje očekávané výsledky.

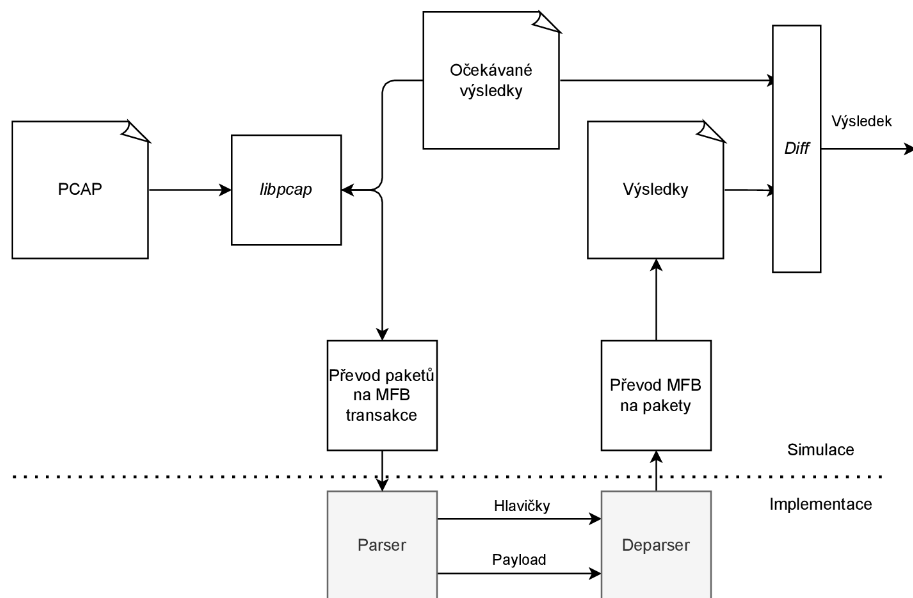
Vytvořené transakce jsou zapisovány do vstupní fronty extraktoru, který je zpracuje a zapíše vyextrahované hodnoty na výstup. Simulace poté vypíše do druhého textového souboru platné hlavičky uvedené ve vektoru příznaků platnosti hlaviček a všechen užitečný náklad až do konce rámce.

Dva vzniklé textové soubory lze porovnat nástrojem *diff*. Pakliže jsou soubory odlišné, simulace končí chybou. Obrázek 4.1 ukazuje schéma takové simulace.



Obrázek 4.1: Schéma simulace extraktoru

Po ověření správné funkčnosti extrakce položek z rámců můžeme přidat i komponentu pro spojení rámců. Díky tomu, že jsme si předem otestovali extrakční blok můžeme předpokládat, že chyba, která se projeví, bude chybou spojování. Výstupem implementace budou tedy spojené rámce uvnitř MFB transakcí. Do textového souboru se budou zapisovat data od značky začátku rámce až po značku konce rámce. Soubory opět porovnáme nástrojem *diff*, čímž zjistíme zda došlo ke správnému sestavení rámců.



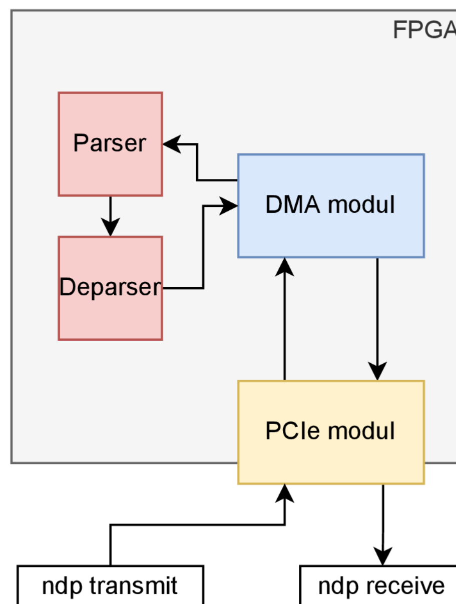
Obrázek 4.2: Schéma simulace komponent

Pro otestování implementace byla použita konfigurace extrakce a spojení zpracovávající protokoly Ethernet, 2x VLAN, IPv4/IPv6, TCP/UDP.

Vstupní soubor typu PCAP může obsahovat libovolné množství rámců. Pro účely testování komponent byly použity zachycené rámce reálného provozu, ale i vygenerované rámce. Zachycené rámce sloužily zejména pro testování velkého objemu dat a byly zachytávány pomocí natroje *tcpdump*. Pro otestování některých krajních případů, jako jsou dvě stejné hlavičky uvnitř vstupního slova a nebo kombinace protokolů, které se moc často v reálném provozu nevyskytují, byl použit jednouduchý skript v jazyce Python využívající knihovnu *Scapy*. Vygenerovaný soubor typu PCAP obsahoval 37 rámců a soubor se zachycenými rámci 5500 rámců.

4.2 Testování v hardwaru

Po ověření funkčnosti v simulacích proběhlo i testování v samotném hardwaru. Aby bylo možné připojit HLS implementaci do NDK platformy vznikla obálka v jazyce VHDL. Tato obálka agreguje vstupní MFB transakce do jednoho vektoru, jenž se pak předává přes rozhraní ap_fifo HLS implementaci. Pro účely testování byla použita karta COMBO-200G2QL. Touto kartou disponuje server s názvem Emilion. Pro funkční testování stačilo propojit příchozí pakety ze softwaru na vstup parseru a výstupní pakety z deparseru na hardwarový vstup DMA. Při takovém zapojení stačí využít dva terminály. První z nich odesílá pakety přes příkaz *ndp-transmit*. Druhý terminál zachytává komunikaci přes DMA příkazem *ndp-receive* a ukládá ji do výstupního souboru typu PCAP. Schéma zapojení můžeme vidět na obrázku 4.3.



Obrázek 4.3: Schéma zapojení pro ověření funkčnosti implementace v hardwaru.

Oba soubory, jak vstupní, tak i výstupní byly zpracovány nástrojem *tcpdump*, jehož výstup byl přesměrován do datového souboru. Nástroj *tcpdump* byl použit s přepínači *-r jmeno_souboru.pcap -xxt*. Přepínač *-r* zajišťuje čtení ze souboru *jmeno_souboru.pcap*. Přepínače *xx* definují, že se má tisknout celý rámec v četně hlavičky linkové vrstvy a přepínač *t* zakazuje výpis časové známky, jelikož v té se budou rámce lišit. Datové soubory lze poté porovnat nástrojem *diff*, jako u simulací. Celkem bylo posláno patnáct paketů s různými

kombinacemi délek a hlaviček. Na obrázku 4.4 můžeme vidět úspěšné přeposlání jednoho paketu.

```
emilion(SL7) ~ $ ndp-transmit -f in.pcap -i 0
------------------------------- NDP transmit stats ----
Packets                    :                    1
Bytes                      :                 1506
Avg speed [Mpps]           :                0.067
Avg speed L1 [Mb/s]        :              816.000
Avg speed L2 [Mb/s]        :              805.333
Time                       :                0.000
emilion(SL7) ~ $ tcpdump -r in.pcap -xxt >in.txt
reading from file in.pcap, link-type EN10MB (Ethernet)
emilion(SL7) ~ $ diff --brief out.txt in.txt
emilion(SL7) ~ $ echo $?
0
emilion(SL7) ~ $
```

```
emilion(SL7) ~ $ndp-receive -R -f out.pcap
-------------------------------- NDP receive stats ----
Packets                    :                    1
Bytes                      :                 1506
Avg speed [Mpps]           :                0.000
Avg speed L1 [Mb/s]        :                0.001
Avg speed L2 [Mb/s]        :                0.001
Time                       :               13.312
emilion(SL7) ~ $ tcpdump -r out.pcap.0 -xxt >out.txt
reading from file out.pcap.0, link-type EN10MB (Ethernet)
emilion(SL7) ~ $
```

Obrázek 4.4: Příjem jednoho rámce bez poškození.

Kapitola 5

Závěr

Hlavní cíl této práce představovala implementace obvodů pro extrakci a spojování hlaviček s využitím vysokoúrovňové syntézy. Aby bylo možné tohoto cíle dosáhnout, bylo nutné si nastudovat problematiku extrakce a spojování, existující řešení a techniky pro syntézu obvodů z vyšších programovacích jazyků. Testování obvodů mělo být provedeno na hardwaru výzkumného týmu Liberouter. Proto bylo nezbytné do studijní části zahrnout i firmware a software dostupný k těmto zařízením. Důležité části nastudované literatury jsou uvedeny v teoretickém úvodu.

Po nastudování literatury jsem se zaměřil na návrh řešení a jeho implementaci. V rámci návrhu jsem usiloval o univerzální řešení pro, co možná nejvíce protokolů. Výsledkem tohoto snažení jsou konfigurovatelné komponenty s využitím C++ šablon nabízející řešení pro protokoly s pevnou délkou hlavičky. S využitím těchto šablon a komponemt pro práci s payloadem, které jsou rovněž výstupem této práce, lze s poměrně malými změnami v kódu vytvářet nové aplikace. Extrakce probíhá bez zpoždění a dokáže zpracovávat i slova sdílená dvěma rámci. Komponenta zpracovává vstupní slova o šířce 512 bitů a pracuje na frekvenci 200 MHz tudíž dosahuje teoretické propustnosti 100 Gb/s. Ovšem během spojování rámců dochází ke zpoždění při rámcích, kde hlavička zasahuje do dvou výstupních slov a tudíž nelze jednoduše odhadnout propustnost této komponenty. Komponenty byly vyvíjeny a testovány v nástroji Vitis HLS 2021.1. Kromě testování uvnitř simulace bylo provedeno i funkční otestování uvnitř hardwaru. Pro tyto účely byla použita karta COMBO-200G2QL. Podrobný popis implementace i testování popisuji v rámci praktické části této práce.

V rámci budoucí práce bych rád vytvořil obálku pro zapojení implementace na ethernetové rozhraní a otestoval implementaci na reálném provozu. Díky tomuto zapojení bude možné změřit propustnost komponent s využitím nástroje Spirent. Dalším krokem by mohlo být připojení části pro akce včetně možnosti konfigurovat pravidla ze softwaru.

Literatura

[1] Ahmadi, M. a Wong, S. Network Processors: Challenges and Trends. In: *In Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2006*. 2006, s. 223–232 [cit. 2022-05-01]. ISBN 90-73461-44-8.

[2] Attig, M. a Brebner, G. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In: *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. 2011, s. 12–23 [cit. 2022-04-28]. DOI: 10.1109/ANCS.2011.12. ISBN 978-1-4577-1454-2.

[3] Bacon, D. F. et al. FPGA Programming for the Masses. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. apr 2013, sv. 56, č. 4, s. 56–63, [cit. 2022-05-01]. DOI: 10.1145/2436256.2436271. ISSN 0001-0782.

[4] Benáček, P., Puš, V. a Kubátová, H. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, s. 148–155 [cit. 2022-04-28]. DOI: 10.1109/FCCM.2016.46. ISBN 978-1-5090-2356-1.

[5] Brebner, G. Packets everywhere: The great opportunity for field programmable technology. In: *2009 International Conference on Field-Programmable Technology*. 2009, s. 1–10 [cit. 2022-04-28]. DOI: 10.1109/FPT.2009.5377604. ISBN 978-1-4244-4375-8.

[6] Cabal, J. *Zpracování síťového provozu na velmi vysokých rychlostech*. 2017. [cit. 2022-04-28]. Diplomová práce. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií.

[7] Cabal, J. et al. Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2018, s. 249–258 [cit. 2022-04-28]. DOI: 10.1145/3174243.3174250. ISBN 978-1-4503-5614-5.

[8] Cabal, J. et al. Scalable P4 Deparser for Speeds Over 100 Gbps. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, s. 323–323 [cit. 2022-04-28]. DOI: 10.1109/FCCM.2019.00064. ISBN 978-1-7281-1131-5.

[9] Coussy, P., Gajski, D. D., Meredith, M. a Takach, A. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers*. 2009, sv. 26, č. 4, s. 8–17. DOI: 10.1109/MDT.2009.69.

[10] De Ghein, L. *MPLS fundamentals.* 1. vyd. Indianapolis: Cisco press, 2007. ISBN 1-58705-197-4.

[11] Efnusheva, D. et al. Memory-centric approach of network processing in a modified RISC-based processing core. In: *2016 Future Technologies Conference (FTC).* 2016, s. 1181–1188. DOI: 10.1109/FTC.2016.7821751. ISBN 978-1-5090-4171-8.

[12] Gibb, G., Varghese, G., Horowitz, M. a McKeown, N. Design principles for packet parsers. In: *Architectures for Networking and Communications Systems.* 2013, s. 13–24 [cit. 2022-05-01]. DOI: 10.1109/ANCS.2013.6665172. ISBN 978-1-4799-1641-2.

[13] Holík, F. a Neradová, S. *Softwarově definované sítě* [online]. 1. Vyd. Univerzita Pardubice, 2019 [cit. 2022-03-20]. ISBN 978-80-7560-235-0.

[14] Ibanez, S. et al. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* New York, NY, USA: Association for Computing Machinery, 2019, s. 1–9 [cit. 2022-04-28]. FPGA '19. DOI: 10.1145/3289602.3293924. ISBN 9781450361378.

[15] Luinaud, T. et al. Design Principles for Packet Deparsers on FPGAs. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* New York, NY, USA: Association for Computing Machinery, 2021, s. 280–286 [cit. 2022-05-01]. FPGA '21. DOI: 10.1145/3431920.3439303. ISBN 9781450382182.

[16] Matoušek, P. *Síťové služby a jejich architektura.* Brno: Publishing house of Brno University of Technology VUTIUM, 2014. 396 s. ISBN 978-80-214-3766-1.

[17] Nane, R. et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* 2016, sv. 35, č. 10, s. 1591–1604, [cit. 2022-05-01]. DOI: 10.1109/TCAD.2015.2513673.

[18] Puš, V., Kekely, L. a Kořenek, J. Design methodology of configurable high performance packet parser for FPGA. In: *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems.* 2014, s. 189–194 [cit. 2022-04-28]. DOI: 10.1109/DDECS.2014.6868788. ISBN 978-1-4799-4558-0.

[19] Satrapa, P. *IPv6* [online]. 4. aktual. a rozš. vyd. Praha: CZ.NIC, 2019 [cit. 2022-03-20]. ISBN 978-80-88168-46-1.

[20] Silva, S. da et al. P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* New York, NY, USA: Association for Computing Machinery, 2018, s. 147–152. FPGA '18. DOI: 10.1145/3174243.3174270. ISBN 9781450356145.

[21] Sosinsky, B. A. *Mistrovství – počítačové sítě.* Vyd. 1. Brno: Computer Press, 2010. Mistrovství (Computer Press). ISBN 978-80-251-3363-7.

[22] CESNET. *Liberouter: Easy to use framework for HW acceleration: Network Development Kit* [online]. [cit. 2022-04-20]. Dostupné z: https://www.liberouter.org/ndk/.

[23] Intel® Tofino™ 2. *Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise* [online]. [cit. 2022-05-07]. Dostupné z: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[24] IEEE Standard for Ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* [online]. IEEE. 2018, s. 1–5600, [cit. 2022-05-01]. DOI: 10.1109/IEEESTD.2018.8457469.

[25] White Paper: Building BittWare's Packet Parser, HLS vs. P4 Implementations. *Bittware: a molex company* [online]. 2021 [cit. 2022-04-08]. Dostupné z: https://www.bittware.com/resources/parser-hls-p4/.

[26] *C/RTL Co-Simulation in Vitis HLS* [online]. 2022 [cit. 2022-05-01]. Dostupné z: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/C/RTL-Co-Simulation-in-Vitis-HLS.

[27] CESNET. *Liberouter: Team* [online]. 2022 [cit. 2022-04-20]. Dostupné z: https://www.liberouter.org/team.

[28] *Pipelining Paradigm* [online]. 2022 [cit. 2022-05-01]. Dostupné z: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Pipelining-Paradigm.

[29] *Using Libraries in Vitis HLS* [online]. 2022 [cit. 2022-05-01]. Dostupné z: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-Libraries-in-Vitis-HLS.