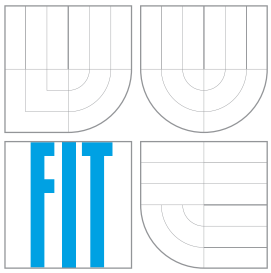
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE NATU A PAKETOVÉHO FILTRU V FPGA PRO 10G SÍTĚ

ACCELERATION OF NAT AND PACKET FILTER IN FPGA FOR 10G NETWORKS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MICHAL ORSÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN VIKTORIN
SUPERVISOR

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Orsák Michal**
Obor: Informační technologie
Téma: **Akcelerace NATu a paketového filtru v FPGA pro 10G sítě**
Acceleration of NAT and Packet Filter in FPGA for 10G Networks
Kategorie: Vestavěné systémy

Pokyny:

1. Seznamte se technologií Kintex-7 od firmy Xilinx, pamětmi QDR SRAM a dostupnými výpočetními moduly s procesory ARM.
2. Nastudujte síťové úlohy a aplikace, které jsou provozovány na hraničních bodech sítí malých ISP (Internet Service Provider) nebo malých/středních podniků. Zaměřte se zejména na problematiku filtrace paketů a překlad IP adres (NAT).
3. Navrhněte systém, který umožní efektivní akceleraci uvedených úloh s využitím technologie FPGA. Předpokládejte, že softwarové zpracování bude probíhat na výpočetních modulech na bázi procesorů ARM.
4. Implementujte systém komponent, který umožní efektivní akceleraci uvedených úloh, a demonstrujte využitelnost komponent na vybrané aplikaci.
5. V závěru diskutujte dosažené výsledky a uveďte možné uplatnění navrženého řešení v různých síťových aplikacích.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

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

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

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

Vedoucí: **Viktorin Jan, Ing.**, UPSY FIT VUT
Konzultant: Puš Viktor, Ing., Ph.D., CESNET
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Cílem této práce je návrh a implementace univerzálního síťového filtrovacího akcelerátoru pro počítačové sítě o rychlosti 10 Gb/s za použití FPGA. Díky přítomnosti pamětí QDR-II může akcelerátor používat značně větší počet pravidel, než by bylo možné za použití vnitřních pamětí FPGA. Vlastnosti akcelerátoru jsou vhodné především pro NAT, paketový filtr a zákonné odposlechy. Platforma, na které filtr pracuje, obsahuje akcelerátor a libovolný počet výpočetních jednotek. Jedna z výpočetních jednotek ovládá akcelerátor prostřednictvím USB, zbytek zpracovává síťový provoz.

Abstract

This thesis deals with the design of a universal hardware acceleration unit for packet filtering in FPGA for 10G networks. Maximum count of rules is greatly increased by the use of external QDR-II memory. Parameters of accelerator are suitable for NAT, packet filtering and lawful interceptions. The platform uses variable number of processing units. One of them controls accelerator by USB port. The rest is used for network processing.

Klíčová slova

NAT, FPGA, síťové filtry, HLS, 10G ethernet, AMBA AXI

Keywords

NAT, FPGA, network filters, HLS, 10G ethernet, AMBA AXI

Citace

Michal Orsák: Akcelerace NATu a paketového filtru v FPGA pro 10G sítě, bakalářská práce, Brno, FIT VUT v Brně, 2016

Akcelerace NATu a paketového filtru v FPGA pro 10G sítě

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Viktorina Další informace mi poskytl Ing. Puš Viktor Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Orsák
18. května 2016

Poděkování

Rád bych poděkoval Ing. Janu Viktorinovi za odborné vedení a hodnotné rady. Poděkování patří také Ing. Viktoru Puši, Ph.D. za informace z reálného prostředí, kde bude zařízení nasazeno. A především celému týmu projektu Sprobe.

Obsah

Kapitola 1

Úvod

Nároky na síťové prvky se zvýší zhruba o 20 % za rok [17]. Pro zpracování provozu na dnešních sítích již nelze používat obyčejné procesory, použití akcelerátoru je tedy nutné. V malých sériích může tento akcelerátorem být realizován síťovým procesorem nebo pomocí FPGA.

Hlavním cílem této práce je navrhnout akcelerátor aplikace NAT a paketového filtru v FPGA. Požadován je škálovatelný systém, který bude schopen zvládat datový tok 10 Gb/s. Současná řešení používají akcelerátory připojené prostřednictvím PCI-Express. Toto rozhraní často není dostupné na úsporných procesorech, zejména ARM a nelze jej za běhu přepojovat mezi výpočetními jednotkami. Řada akcelerátorů je omezena malý množstvím pravidel nebo nízkou propustností. Implementace síťových aplikací jsou bez nízké úrovně abstrakce těžce přestavitelné na jiný případ užití. Flexibilita návrhu tohoto akcelerátoru je zvýšena použitím skriptů v Pythonu a High Level Synthesis (HLS).

Použití Ethernetu pro přenos dat mezi výpočetními jednotkami a akcelerátorem umožní snadné škálování i na rychlostech větších než 10 Gb/s. Ethernet také zjednodušuje fyzické propojování modulů a umožňuje přidávání a odebírání za běhu. Je také téměř na všech dostupných výpočetních modulech. USB 2.0 jako konfigurační rozhraní přináší dostatečnou propustnost, nízkou cenu a je také přítomno na drtivé většině výpočetních modulů.

Rychlé přídavné paměti by pro akcelerátor byly velkým přínosem. Bylo by možné použít daleko více pravidel případně daleko větší vyrovnávací buffery. Díky použití flexibilní architektury, konceptu *Software Defined Monitoring (SDM)* [5] a skriptů bude možné design akcelerátoru snadno konfigurovat a zajistí to jeho lehkou nasaditelnost v jiných aplikacích.

Kapitola 2 popisuje nejčastější úlohy, které se provádí v prostředí, kde bude zařízení použito. Sekce 3.1 a 3.5 popisují nejdůležitější hardwarové technologie, které byly použity. Sekce 4.2 zmiňuje dostupné výpočetní moduly, které se předpokládají pro realizaci výpočetní části systému. V kapitole 5 je návrh systému, ve kterém akcelerátor pracuje, a návrh vlastní akcelerační jednotky. Kapitola 7 ukazuje možnosti modifikace a použití pro jiné účely.

Kapitola 2

Síťové úlohy na hraničních bodech sítí

Standard pro Ethernet 10G (IEEE 802.3ae-2002) je známý již několik let a výrobci jej již implementují v síťových kartách. Malí ISP dnes však nejčastěji používají spoje o rychlostech 1 až 10 Gb/s.

Výjimečná pozice hraničních bodů je výhodná zejména pro služby, které mají mít dopad na všechny pakety směrujících do Internetu nebo do vnitřní sítě. Mezi takové síťové úlohy patří např. Network Address Translation (NAT), firewall, webové cache, legální odposlechy (*lawful interception, LI*) a analýza provozu. Přes hraniční body prochází značná část provozu v síti. To tato zařízení silně zatěžuje. Pro sítě o rychlostech 10 Gb/s (a více) v současné době platí, že pro zpracování veškerého provozu, a to i na nejkratších paketech, vyžaduje hardwarovou akceleraci. Při maximálním zatížení malými pakety má procesor o taktu 3 GHz na zpracování paketu 200 taktů.

2.1 Hraniční body sítí malých ISP

U malých ISP[1] jsou hraniční body nejčastěji tvořeny pomocí hraničních routerů (angl. border router). Díky svému postavení v síti jimi protéká většina toků[2], což tyto prvky výrazně zatěžuje. Toto postavení je však vyžadováno aplikacemi jako je NAT, filtrování paketů, sběr dat apod.

Tyto body jsou jedinými místy, kde je síť poskytovatele připojení připojena k Internetu. Z tohoto důvodu jsou hraniční body primárním místem obrany sítě poskytovatele připojení od okolí v obouch směrech.
Typické aplikace provozované na hraničních bodech jsou:

- *IDS – Intrusion Detection System* je systém, který porovnává aktuální provoz se vzorovou databází provozu. Pokud na základě této databáze detekuje anomálie, ohlašuje incident správci sítě.
- *IPS – Intrusion Prevention System*, pracuje podobně jako IDS. IPS navíc dokáže útoku zabránit.
- Aplikační/paketový firewall.

[1] Internet service provider https://en.wikipedia.org/wiki/Internet_service_provider
[2] datový tok jak jej chápe NetFlow https://en.wikipedia.org/wiki/NetFlow

- Web cache.
- Monitorování provozu.
- NAT –Network Address Translation, popsáno v kapitole 2.3.
- LI – Lawful interception, zákonné odposlechy.

2.2 Filtrace paketů

Linux Kernel, jádro operačního systému Linux, obsahuje algoritmy pro řízení paketového provozu. Tyto algoritmy jsou obvykle implementovány nad frameworkem *NetFilter*[3] a ovládány nástrojem *iptables* z uživatelského prostoru. Jedním z modulů tohoto frameworku je paketový filter.

Paketový filtr v NetFilter využívá 3 tabulky `INPUT, FORWARD, OUTPUT`. Tyto tabulky jsou tvořeny vždy jedním řetězcem (chain) pravidel. Tato pravidla jsou při vyhodnocování procházena sekvenčně. To znamená, že se v nich vyhledává metodou first-match [13].

Sekvenční vyhledávání je vhodné jen pro nízký počet pravidel. Na jedné 10G lince může za sekundu přijít až 14,88 milionu paketů. Procesor má tedy 67.2 ns, aby paket zpracoval, což u procesoru o taktu 3 GHz znamená 200 taktů. Jen přepnutí kontextu procesu trvá minimálně o řád více[4]. Každý příchozí paket navíc způsobí výpadek ve vyrovnávací paměti (cache miss). Nezarovnaný přístup do paměti, přechod mezi uživatelským prostorem (user space) a prostorem jádra (kernel space), obsluha periferií a kopírování bufferů, to vše jsou další problémy, které snižují výkon celého systému. Meziprocesorová komunikace, např. přes QPI[5], má za následek další dramatický pokles výkonu.

V našem případě potřebujeme obsloužit 2×10 Gb/s dat, protože filtrace probíhá v obou směrech. Tento datový tok odpovídá 29,76 Mpps[6]. Hlavní problém je úložiště filtrovacích pravidel a vyhledávání v něm. Aplikace NATu, paketového filtru i LI vyžadují mnohem více vyhledávání než vkládání nových pravidel. Prioritou při vybírání algoritmu, podle kterého bude tato komponenta pracovat, je tedy vyhledávací doba. Základní možnostmi při výběru jsou binární strom a hašování.

Algoritmy založené na binárních stromech a implementované v hardware jsou velmi často vázané na formát vyhledávaného klíče. Na rozdíl od hašování umožňují prefixové vyhledávání ve své výchozí podobě. Algoritmy založené na hašování obvykle nezaručují dobu vyhledávání a plýtvají pamětí, protože tabulka, kterou používají pro uložení záznamů, prakticky není možné zcela zaplnit.

Hašovací tabulka je datová struktura, která pro indexování používá hašovací funkci. Hašovací funkce konvertuje jakkoli dlouhý vstup na výstup pevně stanovené délky (rozsahu). Aby byla funkce použitelná, musí jednotlivé klíče do tabulky mapovat s minimem kolizí. To znamená, že klíče musí být do tabulky mapované co nejvíce rovnoměrně. Je nutné minimalizovat případy, kdy se na stejnou pozici v tabulce namapuje jiný záznam (kolize) [11].

Hašovací schéma zvané kukaččí haš (*cuckoo hashing*) [9] disponuje konstantní dobou vyhledávání. Tato doba je velmi malá a lineárně roste s počtem tabulek. Pokud zanedbáme

[3] http://www.netfilter.org/documentation/index.html

[4] http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

[5] Quick Path Interconnect http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

[6] Million packets per second

kolize, počet pravidel je omezen jen dostupnou pamětí. Kukaččí haš používá několik stejně velkých tabulek a různých hašovacích funkcí. Metoda je implemetovatelná v hardware [12] a lze ji efektivně zřetezit (např. pomocí *d-pipeline* [10]). Tato metoda na základě zmiňovaných vlastností byla vybrána pro implementaci filtru a je jí věnována podsekce 2.2.1.

2.2.1 Kukaččí haš

Kukaččí haš [9] je hašovací schéma, které vyžaduje dvě a více tabulek. Pro jednoduchost dále předpokládejme, že používáme jen dvě tabulky a k nim příslušné funkce.

Při vyhledávání se pro hledaný klíč spočítá haš pomocí funkce první tabulky. Tento haš je indexem v první tabulce. Pokud záznam nalezený na daném indexu odpovídá hledanému klíči, našli jsme hledaný záznam. V opačném případě je stejný postup zopakován v druhé tabulce s druhou hašovací funkcí. Tabulky záznam neobsahují, pokud se ho nepodařilo tímto způsobem najít ani v jedné z nich. Algoritmus je znázorněn v 2.1.

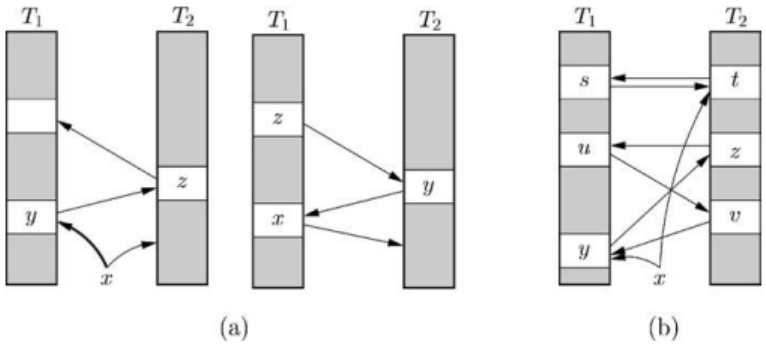
```
def lookup(key):
    for table in tables:
        hashVal = table.hashFn(key)
        element = table[hashVal]
        if element == key:
            # element was found in table on index hashFn(key)
            return element
```

Algoritmus 2.1: Vyhledávání v tabulce.

Při vkládání je záznam vložen do první tabulky. Vkládání je dokončeno, pokud se záznam uložil na místo, které bylo prázdné. V opačném případě je tento původní záznam vložen do následující tabulky, atd. Pokud i po vkládání do poslední tabulky zbude záznam, který se musí uložit do tabulky, je postup opakován od první tabulky. Může se stát, že se algoritmus vkládání zacyklí. Tento případ se dá ošetřit jednoduchým čítačem. Po jeho vypršení je nutné tabulky přeuspořádat a zkusit vložit záznam znovu. Pseudokód algoritmu 2.2 je na obrázku 2.1 graficky znázorněn.

Přeuspořádání je v hardwarové implementaci krajní řešení. Je však možné jej výrazně oddálit použitím přídavné paměti CAM (*Coherent Addressable Memory*, obsahem adresovatelná paměť), do které se uloží daný záznam, pokud se nedá vložit do hlavních tabulek. V praxi se to ukazuje jako nutnost, protože přeuspořádání všech záznamů je časově velmi náročné a dosti často nerealizovatelné. Všechny položky by bylo třeba zkopírovat zpět do procesorové části, protože akcelerátor nemá dostatek paměti na dočasné uložení všech záznamů. A následně by bylo třeba do filtru postupně vložit všechna pravidla, přičemž by mohlo dojít znovu ke kolizi. Filtr by tak byl použitelný jen při malém zaplnění, kde jsou pravděpodobnosti kolizí nízké.

Výskyt kolizí nelze předpokládat. Při nevhodných datech, může velmi snadno dojít ke kolizi i při minimálním zaplnění tabulek. Bez CAM by muselo dojít k přeuspořádání tabulek. Využití CAM lze i snáze monitorovat a tím dostat aproximaci zaplnění tabulek.



Obrázek 2.1: Ukázka vkládání do tabulky s kukaččím hašem. (a) Klíč x je úspěšně vložen místo y a klíče y a z jsou přesunuty mezi tabulkami. (b) Klíč x nemůže být vložen a tabulka musí být přehašována (převzato z [9]).

```
# customizable limit of the atempts to insert to tables before rehashing
maxReinsertion = 10

def insert(key):
    if lookup(key):
        return # key is already stored in tables

    for i in range(maxReinsertion):
        # for each table
        for table in tables:
            hashVal = table.hashFn(key)

            swap(key, table[hashVal])

            if not key.valid:
                # if empty key is found insertion was sucessfull
                return
            # else continue inserting new key which was swapped with previous
     one
    # if maxReinsertion counter was exceeded rehash all tables and try again
    rehash()
    insert(key)
```

Algoritmus 2.2: Vkládání do tabulky.

Při mazání se prohledají všechny tabulky a pokud se záznam v tabulce najde je z ní odstraněn.

2.3 NAT

Technologie *Network Address Tranlaslation* (NAT) vznikla jako přechodná technologie kvůli nedostatku IPv4 adres. Její podstatou je překlad L3 a L4 adres z jednoho adresového prostoru do druhého. Díky tomu je do sítě možno připojit více zařízení, avšak dojde tak k porušení end-to-end[7] principu Internetu. Pojmem NAT Traversal se označuje komunikace se zařízením za NATem [8]. Rozšiřování NAT nejvíce pomohl nedostatek adres IPv4 v Internetu. Další důvod, proč je NAT velmi často nasazován, je ten, že ISP koncovým zákazníkům

[7] https://www.ietf.org/rfc/rfc3724.txt

poskytují pouze jednu IP adresu, zákazníci mají ale často více koncových zařízení, a proto je NAT součástí téměř všech malých routerů. NAT nelze považovat za bezpečností opatření, ale komplikuje skenování sítě za ním ležící.

Technologie NAT se vyskytuje ve více podobách, které se mírně liší svým použitím i funkcí. Všechny tyto modifikace sdílí základní myšlenku, a to překlad adres. Technologie NAT lze dále rozdělit podle více kritérií.

Podle verzí IP adres, se kterými NAT pracuje, se NAT dělí na NAT44, NAT46, NAT444 a podobně. Čísla reprezentují verzi IP protokolu sítě. První reprezentuje síť, ze které se překládá a ostatní do kterých se překládá.

Podle směru překládané adresy se NAT rozděluje na *Source NAT (SNAT) / Destination NAT (DNAT)* . Podle způsobu přiřazování adres na statický a dynamický NAT. Modifikace NATu pro použití ve velkém měřítku se nazývá *Large Scale NAT* (LSN, nebo také Carrier-grade CGN)[8]. Další formou NATu je *Masquerading (MASQ)*, jedná se o formu *Source NAT*, kde je místo adresy nastaven jen výstupní interface. Pro tuto práci je důležité vědět, že je potřeba měnit zdrojové/cílové adresy a porty v paketech.

S NATem souvisí také PAT, obě technologie překládají mezi dvěma adresovými prostory. NAT mapuje adresy 1:1. PAT obvykle používá jedu vnější IP adresu. Uživatelé z vnitřní sítě se připojují do vnější sítě prostřednictvím PAT brány. PAT přidělí stanicím na vnitřní síti porty na své vnější IP. PAT je tedy schopný mapovat v poměru jedna vnější IP na N vnitřních.

V souvislosti s NAT se často objevuje pojem *port forwarding*. Tato technologie používá pevně dána pravidla, která mapují externí dvojici IP a port na interní. Tedy všechna data přicházející na danou IP a port jsou přeposílána na danou interní dvojici bez přičinění jakekoli jiné části NAT.

SNAT je obvykle používán koncovými uživateli internetu, uživatel z vnitřní (např. privátní) sítě inicializuje přenos, tím si v NAT vytvoří pravidlo, které uživateli dočasně přiřadí adresu ve vnější sítí. Z pohledu vnější (např. veřejné) sítě uživatel komunikuje právě z této adresy.

DNAT je možné použít např. pro přerozdělení zátěže na servery. Jedná se o opak SNAT.

Pro NAT obecně platí, že pokud chce spojení inicializovat vnější uživatel (v případě SNAT) není to v základním NAT možné. Využívají se proto následující metody:

- UPnP – Universal Plug-n-Play, technologie, která umožní koncovým zařízením, aby si nakonfigurovala vlastní pravidla pro překlad. Tuto technologii využívají menší domácí routry a je použitelná pro malé domácí sítě, kde nehrozí nepřímé nebezpečí útoku. Tato technologie je totiž snadně zneužitelná.

- STUN[9] a TURN[10] dohromady tvoří ICE technologii. Slouží k navázání spojení mezi dvěma uživateli za různými NATy. Pomocí STUN protokolu se klienti připojí na STUN server, tím se v NATech založí pravidla pro dané uživatele. Pak se za pomocí TURN klienti pokusí připojit napřímo. Tak se dá otevřít P2P spojení i přes NAT, tato metoda se jmenuje NAT Traversal.

[8] http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_nat/configuration/xe-3s/asr1000/nat-xe-3s-asr1k-book/iadnat-cgn.html

[9] RFC 5389

[10] RFC5766, ipv6 RFC6156

Kapitola 3

Použité technologie

3.1 Technologie FPGA

Field Programmable Gate Array (FPGA) je uživatelsky programovatelný integrovaný obvod. FPGA se nejčastěji používají v aplikacích, kde by nasazení ASIC[1] bylo příliš zdlouhavé nebo finančně nevýhodné. FPGA jsou obecně pomalejší než vlastní IC vyrobený stejnou technologií, použití FPGA ale umožňuje implementaci měnit.

FPGA je složeno z rekonfigurovatelných logických bloků, propojovací sítě a dedikovaných bloků. Předními výrobci těchto čipů v roce 2016 jsou Xilinx a Altera. Konfigurovatelné logické bloky jsou velmi často řazeny do sloupců. Větší FPGA bývají rozděleny na hodinové domény, ty největší jsou dále rozděleny na jednotlivé vnitřní čipy, ze kterých je celý čip složen.

U FPGA firmy Xilinx se bloky konfigurovatelné logiky nazývají *Configurable Logic Block (CLB)* [14] a jsou složeny z jednotek *slice*, které obsahují *Look-up Table (LUT)* a registry.

LUT je tvořeno malou SRAM pamětí, multiplexory a registry, čímž poskytuje základ pro implementaci kombinačních i sekvenčních obvodů. Zápisem do SRAM je možné tuto jednotku nakonfigurovat na vykonávání jakékoliv logické funkce.

Dnešní FPGA se obvyklé poskytují následující dedikované bloky:

- IO blocks (IOB),
- paměti RAM (Block RAM), FIFO,
- DSP bloky,
- PCIe bloky, MAC, procesory a další.

Konfigurace pro FPGA je sestavena syntézním nástrojem dodávaným výrobcem. Vstupem do těchto nástrojů je kód napsány v Hardware *Description Langue (HDL)*, např. Verilog, VHDL, SystemVerilog a další. Navrhovat design pro FPGA v jiném jazyce je možné, avšak kód je zpravidla převeden do některého z HDL jazyků, který je následně vysyntetizován.

FPGA umožňují konfigurovat i rozhraní svých fyzických pinů. Pomocí IOB je možné nakonfigurovat typ signálu i jeho napěťovou úroveň. SerDes[2] jednotky (popsáno v sekci 3.4) jsou pevně ukotvené na daný pin pouzdra, ale napěťová úroveň, kódování i protokol jsou konfigurovatelné. Piny pouzdra jsou zpravidla seskupeny do oblastí, které se nazývají *bank*. Tyto oblasti jsou nejmenší jednotkou, na které lze nastavit napěťovou úroveň.

[1] Application-specific integrated circuit
[2] Serializer Deserializer

3.2 Xilinx FPGA

Xilinx[3] je předním výrobce FPGA a pokrývá přibližně 50 % trhu s těmito čipy[4]. Nejnovější modely využívají technologii jako je SSI[5] a 30 Gb/s+ transcievery (viz sekci 3.4). Současně však zaostává za firmou Intel ve výrobním procesu. Sedmá řada Xilinx FPGA[6] používá šesti-vstupé LUT a je aktuálně používanou řadou. Zároveň se v době psaná této práce blíží nástup nové řady Ultrascale a Ultrascale+[7]. Řada 7 FPGA i Ultrascale zahrnuje následující rodiny:

- Atrix - low-end
- Kintex - střední třída
- Virtex - nejpokročilejší
- Zynq - ARM + FPGA

3.3 FPGA Kintex-7

Rodina Xilinx Kintex-7 FPGA[8] je vyráběna 28nm procesem. Čipy poskytují

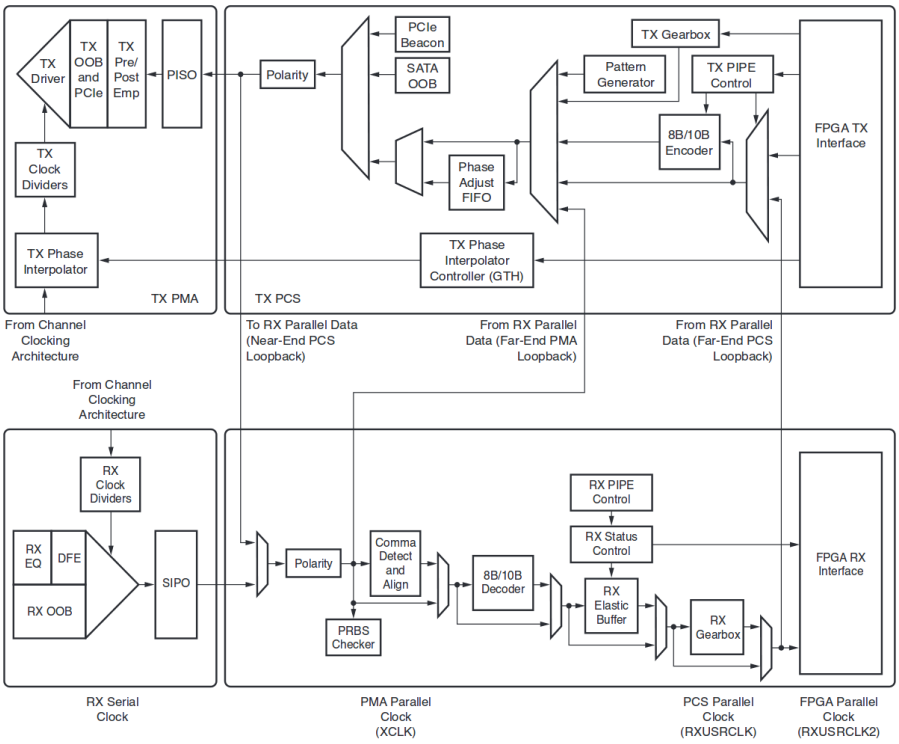
- až 478K logických buněk,
- až 12,5 G Gigabit Transcievers (GTs),
- 2845 GMACs[9],
- 4,25 MB BRAM, DDR3-1866.

Hlavním kritériem při výběru FPGA je dostatečný počet transcieverů, které dokáží přenášet data rychlostí 10 Gb/s. Pro připojení ethernetových portů je klíčové mít dostatek GT (viz sekce 3.4). Každý SFP+ port[10] vyžaduje jeden GT. Celkem jsou tedy potřeba minimálně čtyři 10G transcievery.

Dále je potřeba, aby bylo možné připojit QDR-II, což znamená především dostatek správných pinů i dostatek hodinových domén.

Celkem je potřeba FPGA nejméně Kintex-7 v pouzdru FBG676. Řadič QDR-II pamětí oficiálně podporuje jen větší FPGA, vyžaduje totiž dostatek vysokovýkonnových bank. Při nižších taktech je ale možné provozovat QDR-II i na vysokodosahových bankách. Dále 10G MAC a PCS/PMA jsou oficiální podporovány jen na čipech se speedgrade -2 a vyšším. Nejhorší podporované pouzdro je FFG676. Vybrané FPGA jsou tedy riskantním řešením. Naštěstí je mezi modely zachována pinová kompatibilita, a proto bude případně možné uvedený model vyměnit za jiný.

[3] http://www.xilinx.com/about/company-overview.html
[4] http://www.xilinx.com/about/company-overview.html
[5] Stacked Silicon interconnect, 3D Architectures
[6] http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf
[7] http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf
[8] http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf
[9] Giga MAC(multiply–accumulate operation, $a = a + (b * c)$) per second, pomocí DSP bloků
[10] Small form-factor pluggable transciever



Obrázek 3.1: Schéma Xilinx 7 Series Transceiver (převzato z [15]).

3.4 Gigabit transceivers (GT)

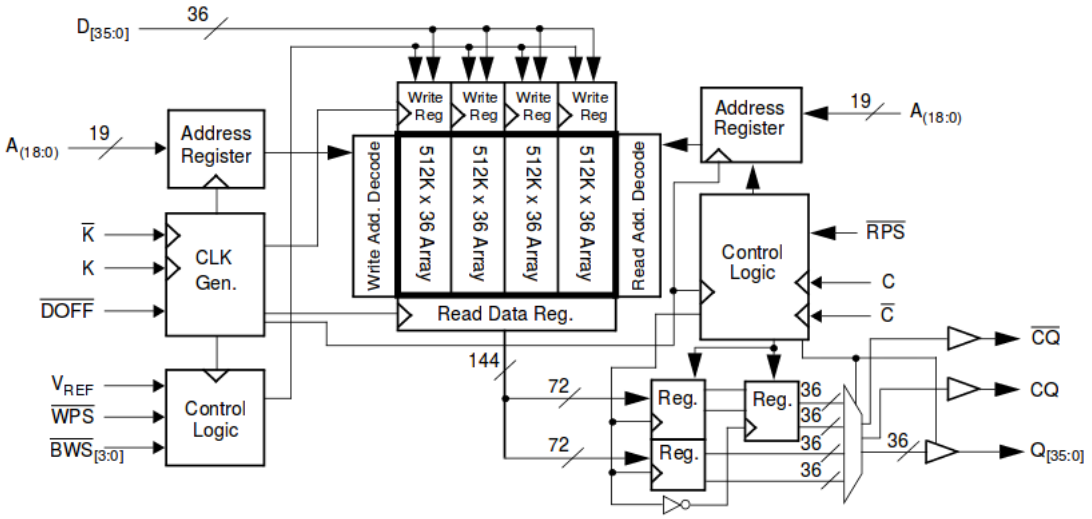
Transceiver je komponenta zajišťující fyzickou vrstvu sériové komunikace. GT jsou tedy transceivery pro seriovou komunikaci v řádu jednotek až desítek Gb/s. Komponenta je rozdělena na *Physical Medium Attachment (PMA)* a *Physical Coding Sublayer (PCS)*.

Jednotka PMA slouží k převodu signálu z vnitřní reprezentace na napeťové úrovně sériového signálu. Vodiče pro komunikaci jsou připojeny na operační zesilovače, které umožňují převod mezi diferenciálním a digitálním signálem. Převod mezi jednobitovým signálem a širším signálem na rozhraní FPGA realizuje serializér, resp. deserializér na přijímací straně. Obecně se tyto jednotky označují pojmem *SerDes*. Jednotky SerDes pro svoji činnost potřebují velmi přesný hodinový signál, ten je generován pomocí jednotky pro fázový závěs PLL nebo přesnější QPLL. Velmi často bývá možné sdílet hodinový signál mezi více transcievery. Umožňuje to značně jednodušší vytváření vícekanálových sériových spojení. U Xilinx 7. řady, UltraScale(+) a u Altera Cyclone V je toto sdílení řešeno další QPLL a hodinovou sítí sdílenou mezi skupinami transceiverů.
Jednotka PCS slouží k dekódování přijatého signálu. Rozhraní do FPGA je většinou připojeno přes asynchronní FIFO.

Vnitřní struktura transceiverů je pro Xilinx 7. řadu znázorněna v obrázku 3.1.

3.5 Paměti QDR-II

Zkratka QDR znamená Quad Data Rate. Jedná se o SRAM paměti, které jsou určeny pro aplikace s požadavky na vysokou propustnost a nízkou latenci. Velmi často nacházejí uplatnění zejména v síťových aplikacích, ve kterých se nejčastěji používají pro uložení bufferů



Obrázek 3.2: Schéma Cypress cy7c1515kv18 (převzato z [2]).

nebo rozsáhlých vyhledávacích tabulek.

Rozhraní tvoří následující signály:

- Vstupní datový signál D.
- Výstupní datový signál Q.
- Adresový signál obvykle značen A, který je sdílen pro oba datové porty.
- Vstupní hodinový signál K pro signál A a D.
- Vstupní hodinový signál CQ pro výstupní datový signál Q.
- Výstupní hodinový signál CQ pro výstupní data Q.
- Signál $\overline{\text{DOFF}}$ (DLL off) ve stavu logické 1 vypne a restartuje PLL nebo DLL jednotu v paměti. To způsobí že se paměť přepne do modu podobného QDR-I a pracuje s latencí 1,5 cyklu.
- Signál $\overline{\text{WPS}}$ (Write Port Select) aktivuje zápis dat ze signálu D.
- Signál $\overline{\text{BWS}}$ (Byte Write Select) aktivuje zápis jednotlivých bytů ve slově.
- Signál $\overline{\text{RPS}}$ (Read Port Select) aktivuje odesílání dat po signálu Q.

Rozhraní se u některých modelů liší, ale princip fungování paměti je vždy stejný.

Datové signály Q a D jsou na sobě zcela nezávislé. Adresa je sdílená pro čtení i zápis. Oba datové porty využívají technologii DDR. Veškeré hodinové signály jsou zpravidla diferenciální a je použit návratový hodinový signál z paměti [3]. Tento signál má frekvenci stejnou jako hlavní hodinový signál, nemá však stejnou fázi. Vysílá jej RAM a řídí přijímací obvody v QDR řadiči. Bez použití tohoto signálu by nebylo možné paměť provozovat na frekvencích v řádu stovek MHz.

QDR-II je přínosná pro aplikace filtru, NAT a LI z následujících důvodů:

Parametry QDR-II	
Frekvence	333 MHz
Náhodných transakcí za sekundu	666 MT/s
Délka slova	9 až 36 b
Latence	1,5 taktu
Napájecí napětí	1,8 V
Délka burstu	2 až 4 slova

Obrázek 3.3: Obecné parametry QDR-II (převzato z [1]).

- Pro všechny aplikace je výhodnější akcelerátor, který dokáže používat více pravidel. Tato pravidla je v FPGA vhodné uložit do BRAM paměti. Tyto paměti mají velmi omezenou kapacitu a jejich počet je omezen, použití této paměti výrazně rozšíří množství pravidel ve filtru. Použitím QDR se spotřebují nepoužité piny a zároveň se uvolní BRAM, které mohou být použity pro implementaci malých bufferů uvnitř FPGA.
- QDR umožní vytvořit buffery o velikostech, které by nebyly v FPGA realizovatelné.

Kapitola 4

Návrh zařízení

Zařízení je rozděleno na akcelerátor a libovolný počet výpočetních jednotek. Obě části spolu komunikují jen pomocí USB za použití protokolu *AxiLite over USB* (viz podsekci 6.2.1) a pomocí Ethernetu za použití *Sprobe10g internal Ethernet* (viz podsekci 6.1.1).

Toto rozdělení podporuje univerzálnost a škálovatelnost výsledné platformy. Výpočetní část může být realizována libovolným zařízením s konektivitou přes Ethernet, nebo i dalším akcelerátorem integrovaným přímo v FPGA. Z tohoto důvodu se o konkrétní implementaci výpočetních jednotek jen okrajově zmiňuje sekce 4.2.

V sekci 2.3 je zmíněno, že algoritmus pro NAT potřebuje neustále vyhledávat stavy ke svým spojením, podobně jako při filtrování. Akcelerátor musí obsahovat úložiště pravidel, ve kterém se pro každý příchozí paket vyhledá pravidlo (popsáno v podsekci 6.7.2), které je pro tento datový tok přednastaveno. Na základě tohoto pravidla je s paketem naloženo. Pokud pravidlo chybí, je provedena předdefinovaná akce. Bez použití akcelerátoru bylo by potřeba vyhradit celé jádro jen na rozřazování paketů do front pro ostatní jádra. Díky tomu, že by každý nový paket způsobí výpadek cache (cache miss), výkon celého procesoru by se razantně snížil. U víceprocesorových desek by se problém ještě zhoršil o kopírování mezi pamětmi jednotlivých CPU.

Zároveň je možné vyhledávání dobře realizovat v FPGA (bylo popsáno v sekci 2.2). Algoritmus vyhledávání pravidla pro daný paket je tedy realizován prostřednictvím akcelerátoru. Akcelerátor obsahuje i jednotku pro aplikování jednoduchých pravidel na paket. Filtrovací jádro pravidlo ve filtrech vždy najde, protože obsahuje jednotku, která umožňuje přenastavit výchozí pravidlo (viz sekci 6.4).

Omezením jsou však implementovatelná pravidla v akcelerátoru. Současná implementace je v sekci 6.7.2 zabývající se pravidlem filtru. K tomuto formátu pravidla je implementována jednotka (viz sekci 6.8), která dané pravidlo aplikuje na rámec. Její implementace je plně dostačující jak pro firewall, tak pro NAT. Návrh je možné vylepšit implementací pravidla, které změní zdrojovou nebo cílovou adresu, případně port, jak je zmíněno v sekci 7.4. Bitová šířka záznamu se tím zvýší o 145 bitů. Toto rozšíření sebou přináší značné komplikace a je proto zařazeno do další fáze vývoje akcelerátoru.

Pro zpřehlednění návrhu jsou dodržovány následující pravidla.

- Oddělení datových cest a řídicí logiky.
- Seskupení souvisejících signálů do rozhraní a jejich standardizace, tím se myslí především maximální používání Axi4-stream a Axi4-lite rozhraní.
- Oddělení jednotky zpracovávající pakety od klasifikačního jádra.

- Standardní postupy při zřetězení.
- Dědičnost jak u rozhraní, tak u komponent.

Pro naši aplikaci byly vybrány paměti od Cypress, konkrétně:

- CY7C1415KV18-300BZXI (1 Mb × 36)[1]
- CY7C1515KV18-300BZXI (2 Mb × 36)[2]

Rozhraní obou pamětí je 36-bitové (z čehož poslední 4 paritní bity nebudou použity), což při 333 MHz umožní teoretickou propustnost 2 × 21 Gb/s. Což je více než dvojnásobně postačující na jednu 10 Gb/s linku (teoreticky). Díky tomu, že adresa je sdílená pro dva kanály, tato propustnost vystačí pro FIFO v obou směrech 10G linky. Tyto paměti byly vybrány s ohledem na cenu. Paměti s 18-bitovou datovou sběrnicí by postačovaly, protože nejsou plánovány buffery pro oba směry linek.

4.1 Statistické vlastnosti L4 filtru

Kolize v hašovacích tabulkách jsou způsobeny nejedinečností klíče. Pravděpodobnost, že k náhodně generovaných celých čísel 0 až N se liší, je dána vztahem [7]:

$$P_{suc.insert} = \frac{N-1}{N} \times \frac{N-2}{N} \times \cdots \times \frac{N-(k-2)}{N} \times \frac{N-(k-1)}{N}$$

Tuto pravděpodobnost aproximuje vztah[3]:

$$P_{suc.insert} = e^{\frac{-k(k-1)}{2N}}$$

Pravděpodobnost výskytu kolize je tedy:

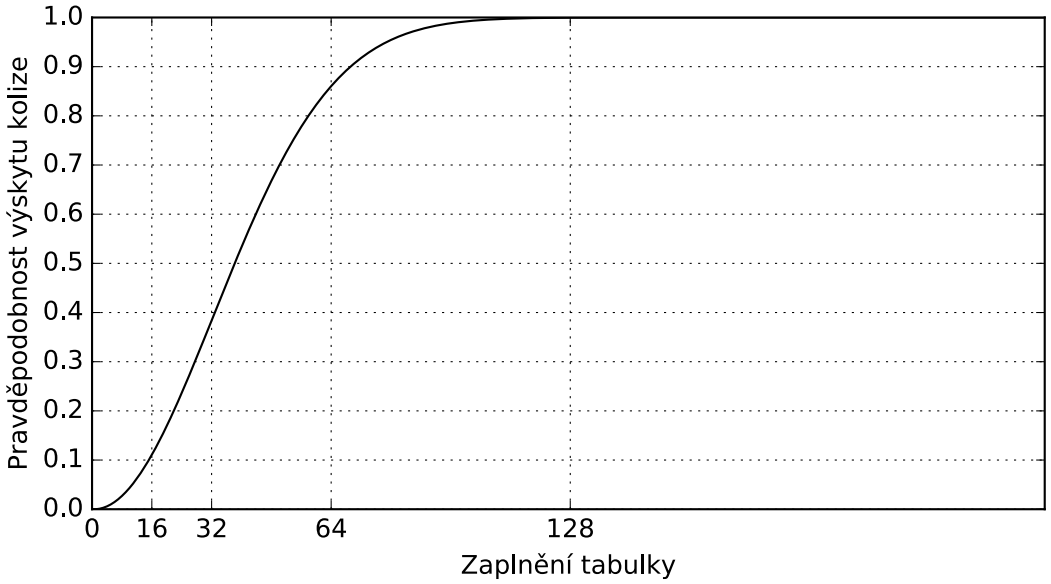
$$P_{collision} = 1 - e^{\frac{-k(k-1)}{2N}}$$

Graf pro tabulku se 1024 prvky je zobrazen v 4.1. Z tohoto grafu je patrné, že pro případ obecné hašovací funkce a obecných dat je zaplnitelná jen minimální část tabulky. Pro přirovnání, ve skupině 23 náhodně vybraných lidí je více než 50 % pravděpodobnost, že nějací dva lidé budou mít narozeniny ve stejný den. Tomuto jevu se v teorii pravděpodobnosti říká narozeninový paradox[4].

Použitím kukaččího haše (viz 2.2.1) se pravděpodobnost výskytu kolizí nezmění. Je tedy patrné, že potenciálně může docházet ke kolizím i při velmi nízkém zaplnění tabulek. Bude tedy výhodné přidat paměť CAM, do které se uloží záznamy, které kvůli kolizím není možné vložit do hlavních tabulek.

Použitím vhodných dat lze ale do tabulek vložit mnohem více, jak ukazují testy filtrovacího jádra (viz 6.3).

[1] http://www.cypress.com/part/cy7c1415kv18-300bzxi
[2] http://www.cypress.com/part/cy7c1515kv18-300bzxi
[3] http://preshing.com/20110504/hash-collision-probabilities/
[4] https://en.wikipedia.org/wiki/Birthday_problem



Obrázek 4.1: Pravděpodobnost výskytu kolize v tabulce o velikosti 1024 prvků.

4.2 Výpočetní moduly

Akcelerátor je použitelný s jakýmkoli výpočetním modulem, který disponuje Ethernetem a USB. Předpokládá se použití těchto modulů:

- Modul ODroid XU4 Samsung Exynos 5422 (4× Cortex-A15 2,0 GHz, Cortexy-A7 1,4 GHz 2 GB LPDDR3, 12 GB/s), 1G Ethernet připojený do USB3.0
- Modul ODroid C1 (4× Cortex®-A5(ARMv7) 1.5 GHz, 1 GB DDR3)
- Mini-ITX s procesory Intel i7 Haswell a novější

Z databází programů pro testování výkonu[5] se ukazuje, že osmi jádrový ARM Cortex-A15 (modul ODroid XU4) bude nejspíš 4 až 5× pomalejší než Intel i7-4770QM.

[5] Antutu http://www.antutu.com/view.shtml?id=7019, http://www.computingcompendium.com/p/arm-vs-intel-benchmarks.html

Kapitola 5

Navržený systém

Systém, který popisuje tato kapitola, prošel několika iteracemi. V prvotním řešení měl měl být použit větší počet samostatných mikropočítačů osazených procesory ARM, které na sobě měly být zcela nezávislé. Všechny výpočetní jednotky komunikovaly jen prostřednictvím Ethernetu. Konfigurace filtru tedy probíhala pomocí paketů zasílaných na akcelerátor prostřednictvím stejné sítě jako data. Akcelerátor měl obsahovat oddělené filtrační jádro pro každý výpočetní modul. Systém podporoval dynamické přidávání a ubírání výpočetních jednotek.

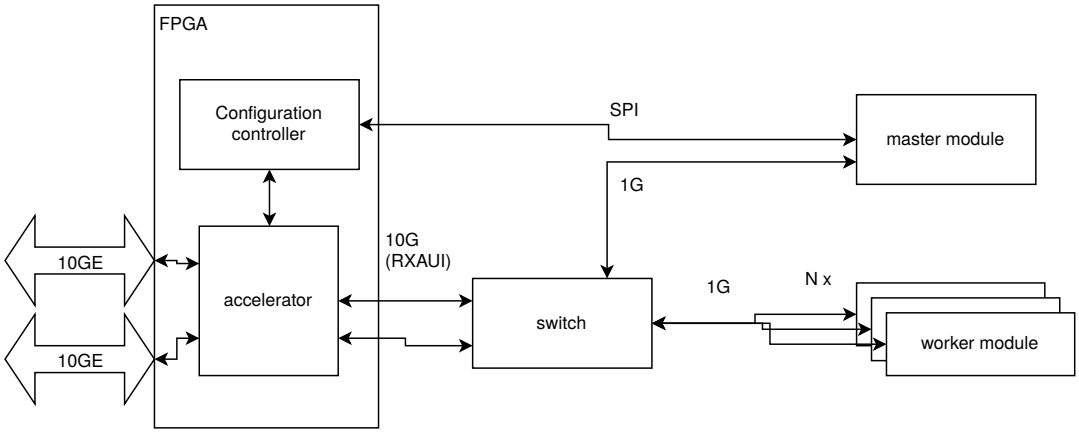
Řešení se ukázalo nerealizovatelné se současnými FPGA v dané cenové relaci. Celá platforma byla taktéž v konečném důsledku velmi komplikovaná. Hlavním cílem bylo však udělat výpočetní moduly na sobě nezávislé a tím systém zjednodušit. Při podrobnější analýze se ukázalo, že výsledná implementace nemůže obsahovat nezávislé jádra a že informace mezi nimi budou muset být synchronizovány. Zároveň zdroje FPGA nestačily na instance více filtrů a rozdělení tabulek filtru pro dané moduly s sebou přineslo výrazné zpoždění. Jako rozumně nerealizovatelná se taktéž ukázala možnost dynamicky rozdělovat dostupnou paměť do tabulek podle počtu výpočetních jednotek. Toto řešení bylo později označeno jako verze 0.

Kvůli tomu, že systém verze 0 nebyl implementovatelný, přešlo se k verzi 1, která je popsána v sekci 5.1. Tento návrh byl dále zoptimalizován použitím USB místo SPI a odstraněním switche z hlavní desky akcelerátoru. Platforma tak byla přepracována na verzi 2 (sekce 5.2). Tato platforma nemusí používat switch. Je možné připojit jednu výpočetní jednotku, která má dva 10G porty. Tato vlastnost je vhodná pro desky s výkonnějšími x86-64 procesory. Současně škálovatelnost je zachována, protože switch je pořád možno připojit. A akcelerátor tak lze použít i v konfiguraci s ARM procesory nebo v jakékoli kombinaci.

5.1 Verze 1 – multiprocesorový systém s moduly ARM

Výpočetní moduly jsou připojeny na centrální switch prostřednictvím 1G Ethernetu. Jeden z modulů zastává funkci řídicí jednotky (dále jen master). Tato jednotka obstarává údržbu akcelerátoru a všechny ostatní konfigurují akcelerátor skrze ní. Vlastní akcelerátor je k akcelerační platformě připojen skrz switch dvěma ethernetovými porty (RXAUI). Veškerá komunikace mezi jednotlivými částmi systému se děje prostřednictvím switche. Vnitřní síť používá L2 adresování, tedy akcelerátor se nemusí zabývat daty ethernetových rámců a pro směrování stačí jen upravit adresy v ethernetovém rámci.

- Chce-li výpočetní modul přijímat datové toky, musí se nejprve zaregistrovat skrze



Obrázek 5.1: Propojení systému v první verzi platformy.

hlavní jednotku.

- Po registraci akcelerátor směřuje na modul část síťového toku.
- Modul může přes hlavní jednotku konfigurovat filtr, tedy může se z toku odhlásit, zaregistrovat se do toku, který ještě nemusí neexistovat, apod.

Tuto platformu znázorněnou na obrázku 5.1 tedy tvoří následující logické celky:

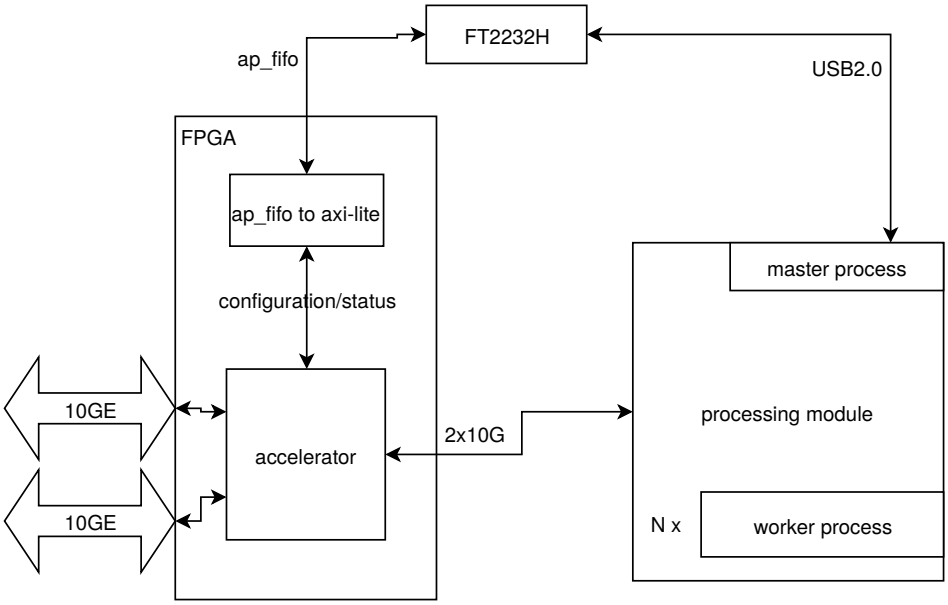
- Výpočetní jednotka ovládající filtr v FPGA (master).
- Aplikační jednotky (worker), které jsou reprezentovány jádry výpočetních modulů a plní libovolnou úlohu nad síťovými toky,
- Akcelerátor filtru, kombinovaný filtr v FPGA, který zároveň směruje pakety ke správné aplikační jednotce, řízen hlavní jednotkou.

Základní deska akcelerátoru obsahuje

- úložiště pro uložení konfigurace FPGA,
- QDR-II paměti,
- dvě klece pro SPF+ moduly,
- ethernetový switch pro komunikaci s výpočetními moduly,
- a další elektroniku.

Celkově hardware poskytuje 2 SFP+ porty a až 24 1 Gb/s portů pro připojení výpočetních jednotek. Hlavní modul ovládá akcelerátor pomocí SPI[1].

[1]Serial Peripheral Interface https://cs.wikipedia.org/wiki/Serial_Peripheral_Interface



Obrázek 5.2: Propojení systému poslední verze (2).

5.2 Verze 2 – systém s procesorem Intel i7

Deska akcelerátoru má 4 SFP+ porty. Dva slouží k připojení vnější sítě a zbylé k připojení výpočetního modulu, který je v základní verzi realizován standardní deskou s procesorem z rodiny Intel i7. QDR paměti jsou v této verzi volitelné a jsou volitelně použity na vyrovnávací buffery nebo tabulky filtru. Konfigurace akcelerátoru probíhá přes USB. Deska akcelerátoru obsahuje převodník (FT2232H) USB 2.0 na FIFO rozhraní, jak je zaznačeno ve schématu 5.2. Toto rozhraní je přivedeno do FPGA. FT2232H má i další rozhraní, na které je připojena flash paměť a generátor hodin.

V architektuře jsou tedy následující logické celky:

- Hlavní jednotka (master) je proces, který ovládá filtr v FPGA.
- Aplikační jednotký (worker), které jsou v tomto případě reprezentovány procesy. Filtr na ně směřuje data a aplikační jednotka nad nimi provádí požadované úkony, chápejme aplikační jednotku jako instanci síťové aplikace
- Akcelerátor filtru zůstal stejný jako ve verzi 1, změny zaznamenaly jen způsoby připojení a komponenty na těchto rozhraních.

Kapitola 6

Akcelerátor v FPGA

V této kapitole je popisován výhradně akcelerátor poslední verze (sekce 5.2). Akcelerátor je rozdělen na filtrovací jádro, paketové procesory a periferie, jak je vidět na obrázku 6.1.

Jako konfigurační sběrnice je použita sběrnice AMBA AXI4-Lite[1]. Tato sběrnice je standardem v projektech pro FPGA s integrovaným procesorem ARM, u ostatních FPGA je v dnešní době hojně používána. Na servisní sběrnici je jediné zařízení typu master, a tím je převodník na obousměrné FIFO rozhraní. Toto rozhraní je vyvedeno ven z FPGA a je na něj připojen převodník na USB, přes který je akcelerátor konfigurován.

Pro zpřehlednění nejsou zmiňovány adresové mapy ani konkrétní registry. Tyto parametry podléhají konfiguraci a jsou popsané v hlavním souboru každé jednotky. Dále pro zjednodušení nejsou zmiňovány buffery, redukce, jednotky typu fork (převod streamu 1:N), binder (převod streamu N:1), crossbar (převod streamu N:N), drop (pro zahazování rámců/slov), apod.

Ethernetové vstupy jsou po vstupu napojeny na PCS/PMA a MAC pro 10G Ethernet, které tvoří Ethernet subsystém[2]. Tyto komponenty jsou připojeny na servisní sběrnici. Datové spojení s filtrem zajišťují vždy dvě Axi4-stream rozhraní 6.1.

V systému můžeme pozorovat dva typy cest pro pakety.

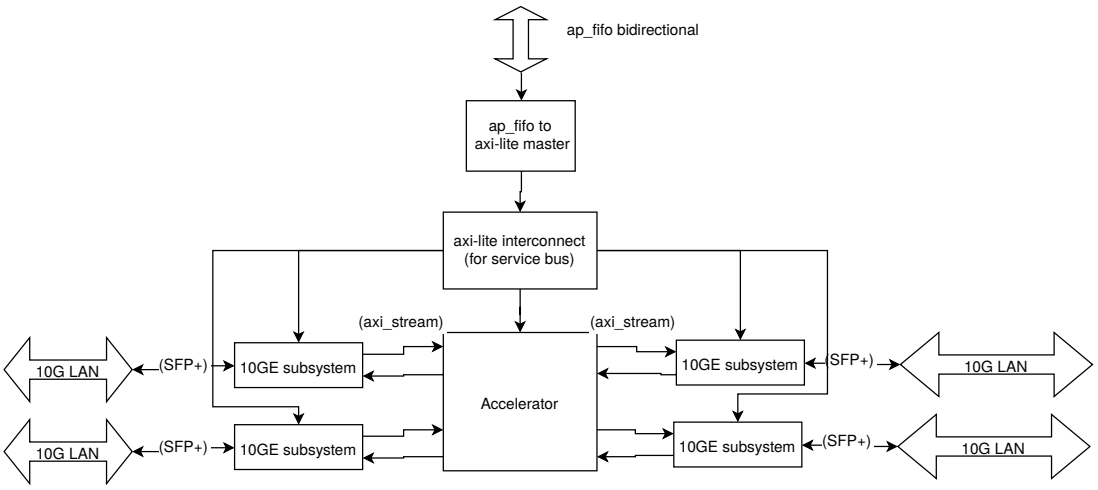
Cesta paketů do systému z vnějších rozhraní začíná extrahováním informací o paketu pomocí komponenty *Head Field Extractor (HFE)* a tato data jsou pak předávány filtrovacímu jádru. Paket samotný zároveň putuje do jednotky *Packet Internal Router*, kde je zahozen, přeposlán na procesorovou jednotku nebo přepojen na opačný port podle rozhodnutí filtru.

Výstupy z HFE jednotky jsou protokol, zdrojová a cílová adresa, port a verze IP adres. Na základě těchto informací filtrovací jádro rozhoduje. Jednotka `Filter result to routing cmd` je překladač mezi výsledkem filtru a routovacím pravidlem. Její význam spočívá v oddělení logiky směrovačů ve filtru od vlastního filtrovacího jádra. Směrování na procesorovou jednotku se děje pomocí interního ethernetového protokolu 6.1.1 paketu a odeslání na port vedoucí k desce s CPU.

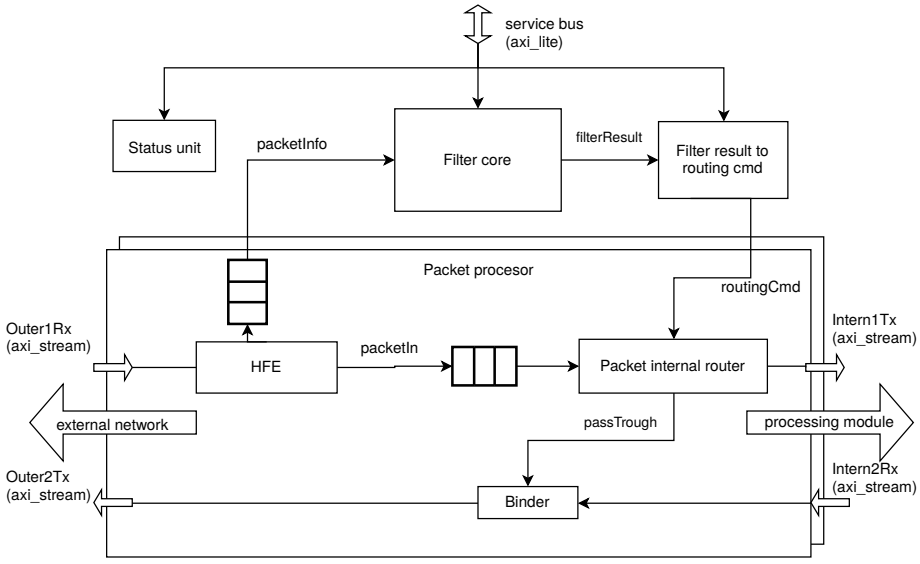
Cesta paketů ze systému ve filtru neni kontolována. Datový tok je jen sloučen s pakety, které byly filtrovacím jádrem určeny přímo k propuštění na vnější síť (rozhraní *passTrought*). Výstup z filtru je napojen přímo na výstupní Ethernet subsystém a je tedy přeposílán přímo na výstupní SFP+ port.

[1] http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

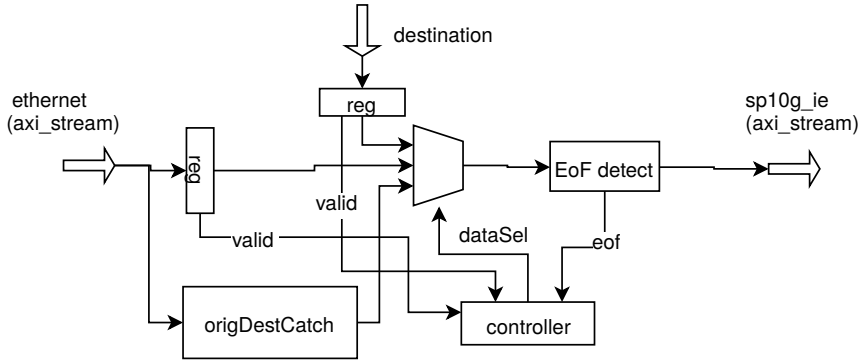
[2] http://www.xilinx.com/support/documentation/ip_documentation/axi_10g_ethernet/v2_0/pg157-axi-10g-ethernet.pdf



Obrázek 6.1: Základní struktura akcelerátoru.



Obrázek 6.2: Základní struktura filtru. Hodní a spodní axi-stream kanál jsou každý z opačného eternetového portu.



Obrázek 6.3: Struktura jednotky pro převod Ethernetu na Interní ethernetový protokol.

6.1 Jednotka pro převod Ethernetu na Interní ethernetový protokol

Převod rámce Ethernetu na rámec Interního Ethernetu je realizován vložením nové cílové adresy a uložením původní do datové části rámce. Tato změna je ukázána v sekci 6.1.1. Rozhraní destination je handshakované a nese novou cílovou MAC adresu.

6.1.1 Interní ethernetový protokol pro Sprobe10g

Jedná se o protokol umožňující transport ethernetových rámců z vnější sítě na interní síti mezi akcelerátorem a výpočetními moduly, při zachování původní cílové adresy. Jedná se o standardní Ethernet 6.4 s jedinou výjimkou. Do dat (payload) je na začátek přidáno 8 bytů, z čehož první dva jsou nevyužité a zbylých 6 je původní cílová MAC adresa. Tato původní adresa reprezentuje originální cílovou MAC adresu z původního rámce přijatého akcelerátorem. Cílová adresa v rámci je použita pro směrování na výpočetní moduly originální cílová adresa je použita pro obnovení původního tvaru rámce při odesílání do vnější sítě. Původní zdrojová adresa zůstává zachována, stejně jako zbytek původního rámce, jak je vidět na obrázku 6.5 (původní type/size je taktéž zachován).

Nutno připomenout, že na interních axi-stream sběrnicích se vyskytují rámce bez preambule, oddělovače začátku rámce i kontrolního součtu [16].

7B	1B	6B	6B	2B	0-1500B	0-46B	4B
Preambule	Start of Frame Delimiter (SFD)	Destination Address	Source Adress	Length/Type	Data	Pad	FCS

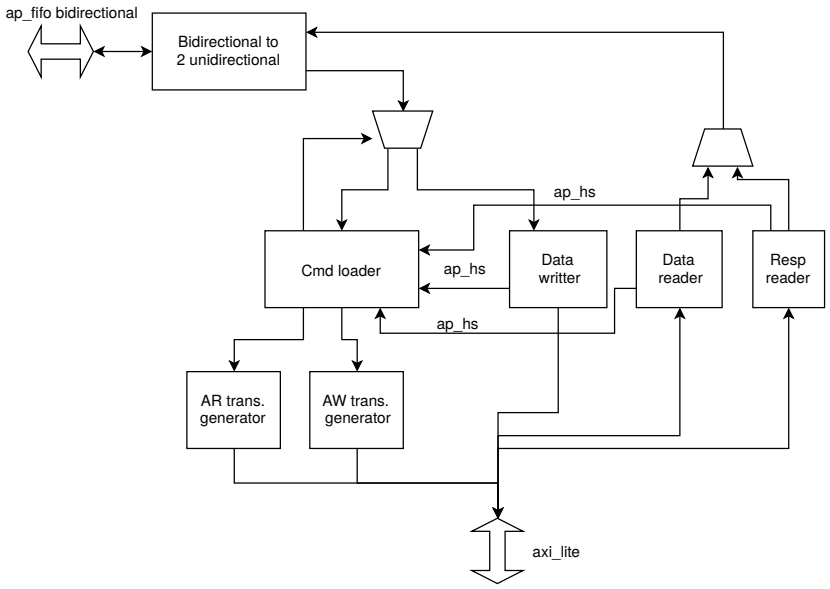
Obrázek 6.4: Formát IEEE 802.3 Ethernet II rámce.

7B	1B	6B	6B	2B	2B	6B	0-1500B	0-46B	4B
Preambule	Start of Frame Delimiter (SFD)	Destination Address	Source Adress	Length/Type	Not Used	Original Dest. addr.	Data	Pad	FCS

Obrázek 6.5: Formát rámce SP10InternalEthernet.

Pro tento protokol se předpokládá následující použití:

- Akcelerátor v rámci směrování na výpočetní moduly zabaluje ethernetové rámce z vnější sítě pomocí tohoto protokolu. Jako cílová adresa je použita adresa výpočetní jednotky a cílová adresa z původního paketu je uložena do originální cílové adresy.



Obrázek 6.6: Struktura jednotky packet internal router.

- Po příchodu rámce na výpočetní jednotku je rámec přiřazen příslušnému procesu ovladačem síťové karty. Dále je přepsána cílová adresa originální cílovou adresou, následně je celá hlavička posunuta o 8B směrem ke konci paketu. Tím vznikne originální rámec.
- Do procesu, který provádí danou operaci nad takovým tokem se tedy dostane původní ethernetový rámec, který přišel na akcelerátor. (Kontrolní součet je zkontrolován hardwarem síťové karty výpočetního modulu a po vstupu do software již není potřeba[3].)

6.2 Jednotka pro převod USB na Axi4-Lite

Tato komponenta slouží pro připojení USB na interní servisní sběrnici. Struktura je znázorněna ve schématu 6.6.

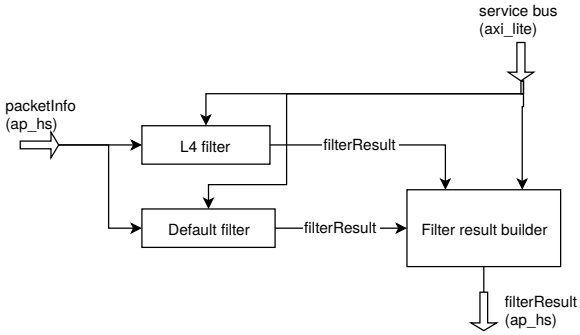
Kanál pro čtení, zápis i potvrzení o zápisu jsou spolu příliš provázané kvůli obousměrnému FIFO rozhraní směrem k USB. Ze strany USB komponenta komunikuje protokolem popsaným v podsekci 6.2.1.

6.2.1 AxiLite over USB FIFO

Axi4-Lite v obvyklých designech používá víc než 128 bitů. Rozhraní s tolika bity lze v reálných případech použít jen uvnitř čipu. Žádný převodník z USB přímo Axi4-lite neexistuje jako integrovaný obvod. V rámci akcelerátoru je však Axi4-lite použita jako servisní sběrnice. Řídicí jednotka vyžaduje přístup na servisní sběrnici.

V zařízení je použit čip FTDI FT2232H[4]. Pro tento protokol je podstatné, že čip převádí USB na rozhraní podobné obousměrnému FIFO rozhraní. Toto rozhraní je zjednodušeně poloduplexní stream, který nepodporuje signalizace rámců. Jedná se o formu obousměrného FIFO rozhraní, kde jsou datové vodiče sdíleny pro oba směry. Byla naimplementována

[3] Pakety posílané z výpočetních jednotek na akcelerátor používají standardní ethernet.
[4] http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf



Obrázek 6.7: Struktura filtrovacího jádra.

komponenta, která umožňuje převod tohoto rozhraní na Axi4-Lite, která využívá tento protokol.

Protokol samotný používá rámce variabilní délky i potvrzování zápisu, které si aplikace vyžadují. Obě strany si musí uchovávat stav komunikace.

Formát požadavkového rámce:

Bit	0 až 6	7	8 až 39	zbytek
Položka	délka v bajtech	r/w (w=1)	32 b adresa	případná data pro zápis

Potvrzení o zápisu je hodnota z posledního Axi4-Lite potvrzení o zápisu rozšířená na 1 byte. To znamená 0 pro úspěch a 2 pro neúspěch. Čtecí data jsou odesílána bez jakékoli přidané hlavičky.

6.3 Filtrovací jádro

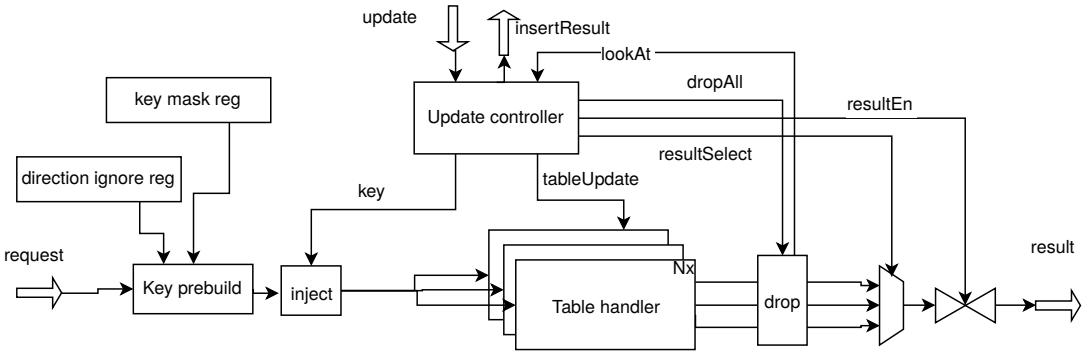
Struktura této komponenty je znázorněna na obrázku 6.7. Jádro je složeno z několika filtrů aktivní jsou vždy všechny. Komponenta *Filter result builder* slouží k vybrání filtrovacího pravidla s nejvyšší prioritou. Výsledkem této komponenty je jediné pravidlo, které vystupuje přes rozhraní *filterResult* a ovládá komponentu pro přeposílání rámců (viz v sekci 6.8). Rozhraní všech filtrů je stejné, přidání jakéhokoli dalšího je triviální záležitost.

6.4 Výchozí filter

Jednotka, jejíž výstup je nezávislý na vstupu. Výstup je konfigurovatelný přes servisní rozhraní. Komponentu lze nakonfigurovat do následujících režimů.

- Generuje pravidlo `ROUTE` $(1 \ll N)$, kde N je hodnota haše informací o paketu modulo počet procesorových jednotek.
- Generuje pravidlo `ROUTE` $(1 \ll N)$, kde N je hodnota kruhového čítače zvětšujícího se při každé klasifikaci.
- Na výstup posílá hodnotu nakonfigurovanou přes servisní rozhraní.

Po restartu je komponenta v prvním režimu, tedy rovnoměrně rozkládá pakety na výpočetní moduly pomocí hašovací funkce. Tato komponenta je nutnou součástí, v případě, že by filtrovací jádro nenalezlo pro daný paket pravidlo, došlo by k zamrznutí celého systému.



Obrázek 6.8: Struktura paralelní implementace L4 filtru.

6.5 Paralelní L4 filtr

Tento filtr používá vyhledávací tabulky s kukaččím hašem. Přičemž díky konfigurovatelné maskovací jednotce je schopen pracovat s různými položkami klíče. Tato komponenta je v implementaci pojmenována jako `sp10g.filterCore.l4filterParallel.core`, do VHDL se převádí jako `L4filterParallelCore`. Samotná komponenta `L4filterParallel` je obálka pro tuto komponentu, která obsahuje navíc registry připojitelné na servisní rozhraní a převodník z těchto registrů na update rozhraní tohoto filtru. Tato sekce se zabývá jen komponentou `L4filterParallelCore`.

Počet tabulek je konfigurovatelný, stejně jako počet jejich položek i počet položek v CAM. Pro CAM je vytvořena obálka `CAM handler`, která umožňuje přistupovat k CAM stejně jako k ostatním tabulkám.

Na rozhraní `request`, které je odvozeno od třídy *PacketInfoWithChannel* (handshakované rozhraní, signál pro kanál a klíč filtru), filtr naslouchá pro příchozí požadavky na vyhledávání. Po příjetí požadavku jednotka *Key prebuild* sestaví z informací o paketu klíč. Tento klíč je následně vyhledán ve všech tabulkách. Záznam se v tabulkách může najít maximálně jednou. Na výstup tohoto filtru je přeposlán nalezený výsledek nebo výsledek z první tabulky (tzn. nenalezeno).

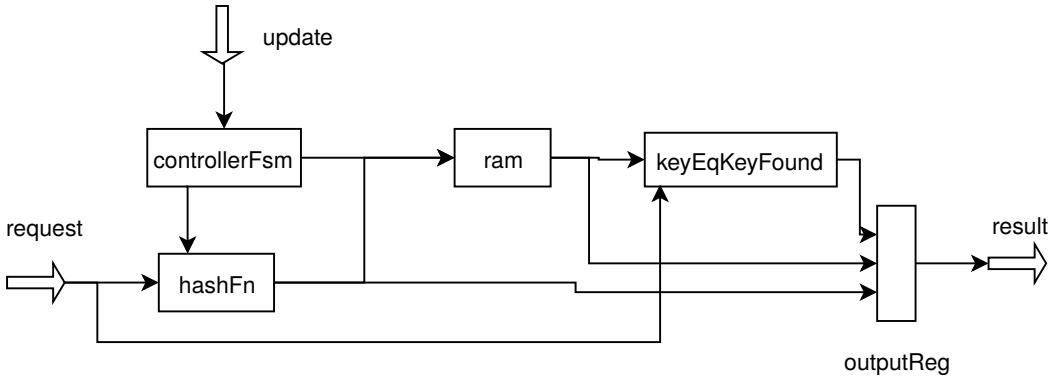
Jednotka obsahuje kontrolér, který se stará o aktualizaci tabulek. Tato jednotka je popsána v podsekci 6.5.3.

6.5.1 TableHandler

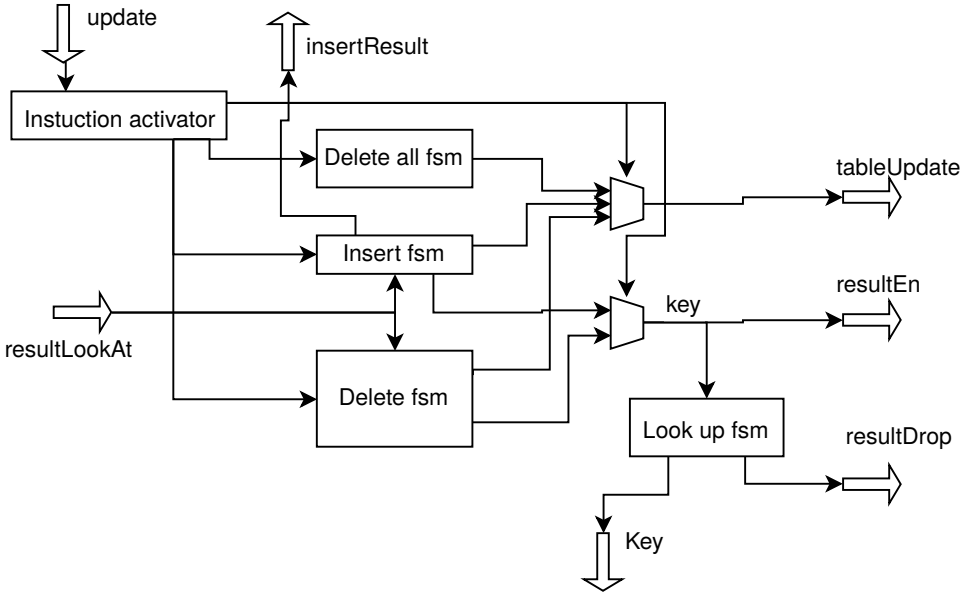
Tato jednotka obstarává vyhledávání v tabulkách s kukaččím hašováním. Obsahuje vlastní tabulku uloženou v jednoportové paměti *Block RAM*, hašovací funkci, kontrolní jednotku aktualizace a výstupní datový registr, jak je znázorněno na obrázku 6.9. Vstupem je klíč, výstupem nalezený záznam, pozice v tabulce a příznak nalezení hledané položky.

6.5.2 CAMHandler

V rámci této práce byla použita existující implementace paměti CAM, která je mapována přímo na LUT v FPGA. Obsahuje port pro vyhledávání a aktualizaci. V CAM jsou uloženy jen klíče, pravidla a příznaky validity jsou uložena v interních registrech této komponenty. Tato komponenta obsahuje obvody, které zaručují aktualizace a vyhledávání v CAM a tím i interních registrech. Komponenta komunikuje s okolím za použití stejného rozhraní, jako komponenta *TableHandler* (viz podsekci 6.5.1), ve filtru ve kterém je použita se tedy chová



Obrázek 6.9: Struktura jednotky TableHandler pro paralelní implementaci L4 filtru.



Obrázek 6.10: Struktura kontrolní jednotky pro aktualizování v paralelní implementaci L4 filtru.

stejně jako hašovací tabulky. Jediný rozdíl je, že při nenalezení daného záznamu nastavuje index v rozhraní `result` na index prvního volného místa.

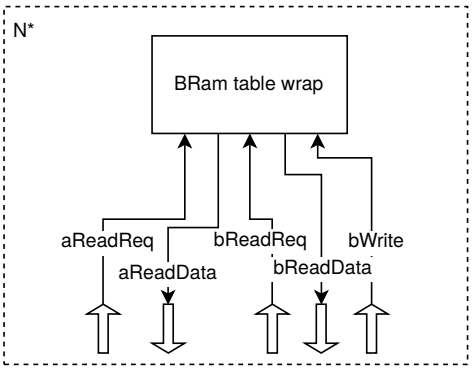
6.5.3 Controller

Tato jednotka je příliš spjata s jádrem paralelního filtru popsaného v sekci 6.5 na to, aby její implementace mohla být samostatná komponenta, je tedy implementována v rámci jádra paralelního L4 filtru. Komponenta řídí vkládání a mazání jednotlivých pravidel. Přes rozhraní `update` lze zapnout i inicializaci tabulky, to znamená smazání všech prvků. Při vkládání jednotka odesílá přes rozhraní `insertResult` příznak úspěchu. V případě neúspěchu přes toto rozhraní odesílá i záznam, který už není možné nadále uchovat v tabulkách. Komponenta má simulační model na úrovní streamů (`L4filterCoreStreamLvl`). Implementaci tohoto modelu lze nalézt v balíčku `sp10g.filterCore.l4FilterParallel.filterCoreStreamLvl` .

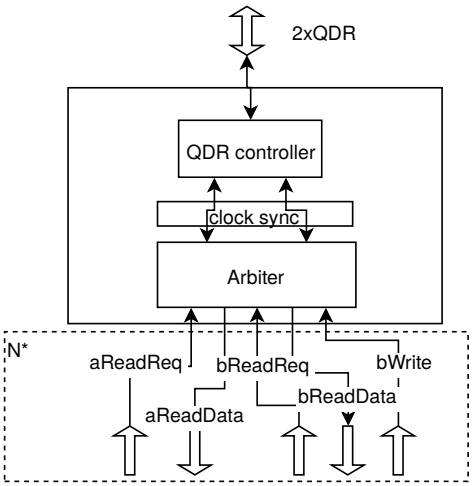
6.6 Zřetězený L4 filtr

Tento filtr používá stejný vyhledávací algoritmus jako paralelní verze v sekci 6.5. Na rozdíl od paralelní verze, je implementován jako soustava zřetězených linek, pro dosažení maximální propustnosti. Filtr je složen ze čtyř hlavních částí, jak je vidět na obrázku 6.15.

Memory with arbiter je jednotka, která zastřešuje implementace vyhledávacích tabulek v tomto filtru. Umožňuje jednoduše zakrýt rozdíl mezi Block RAM implementací a implementací používajících paměti QDR. Poskytuje několik rozhraní využívajících handshake synchronizaci, jak ukazuje schema 6.6. V základní verzi je pro každou tabulku použita jedna komponenta `Memory with arbiter`, QDR implementace má znásobené rozhraní.



Obrázek 6.11: Implementace jednotky `Memory with arbiter` s použitím pamětí Block RAM.



Obrázek 6.12: Implementace jednotky `Memory with arbiter` s použitím QDR řadiče.

Rule CAM je obálka pro *Coherent Adresable Memory* (CAM) s přídavnými registry pro pravidla a arbitrací přístupu. Klíč je stejný jako vyhledávaný prvek v hašovacích tabulkách. Důvody použití CAM jsou zmíněny v sekci 6.5.

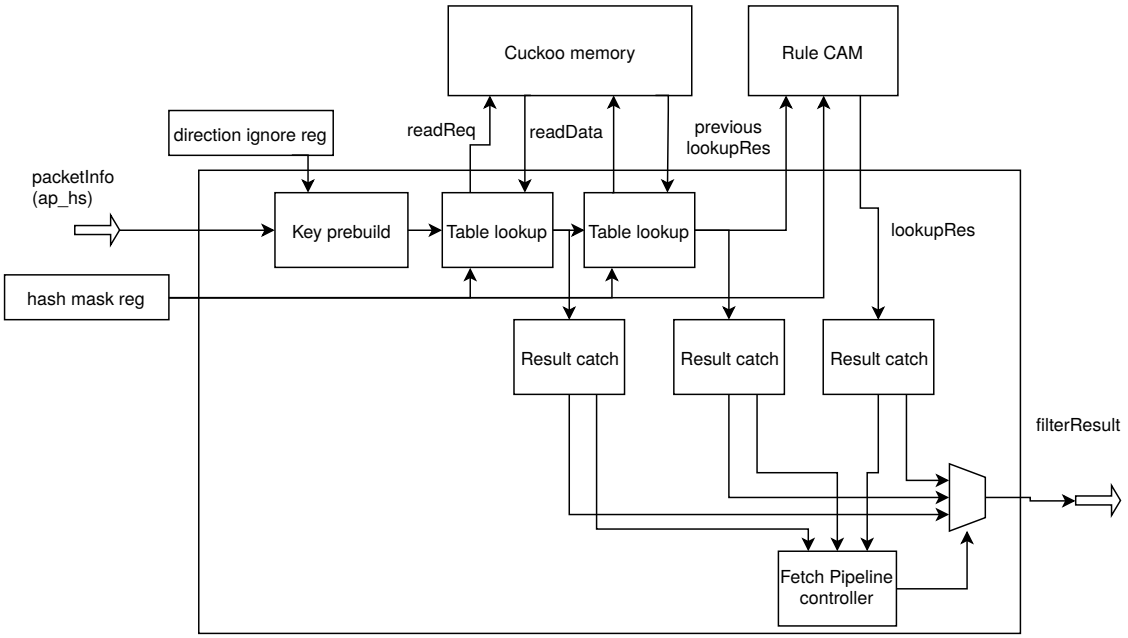
Vyhledávací jednotka (*Fetch unit*) realizuje vyhledávací algoritmus nad tabulkami a CAM. Podobně jako paralelní verze 6.5, komponenta obsahuje jednotku pro předpracování klíče. Jednotky pro vyhledávání v tabulkách jsou zřetězeny a pracuje s nimi linka, která na výstup filtru posílá vždy záznam nalezený v tabulce s nejvyšším indexem, případně záznam z CAM.

Vyhledávací jednotka (Table lookup) obsahuje hašovací jednotku, jednotku pro načítání z připojené paměti a kontrolní jednotku, která řídí aktualizace tabulky a povoluje vyhledávání. Aktualizační (update) jednotka je ovládaná prostřednictvím registrů připojených na servisní sběrnici. Používá algoritmus 2.2. V případě, že přeteče počítadlo pokusů vkládání je pravidlo vloženo do rule CAM. Pokud je i rule CAM zaplněná vrací se pomocí axilite kanálu pro potvrzení zápisu (B) hodnota značící chybu (SLVERR).

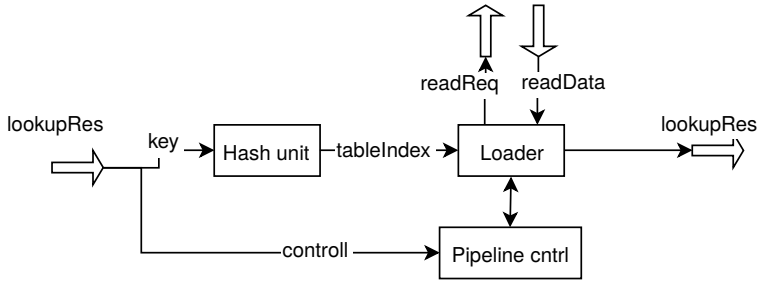
Horní hranice latence filtru L_{hF} je dána jako:

$$L_{hF} = |T| \cdot 2 + |CAM| + 1$$

Kde *CAM* značí počet položek paměti CAM a T počet použitých tabulek. Spodní hranice je 3 takty. Zpoždění filtru je 1 až `|CAM|` $+ 1$ taktu.



Obrázek 6.13: Struktura vyhledávací jednotky pro zřetězenou implementace L4 filtru (2 tabulky).

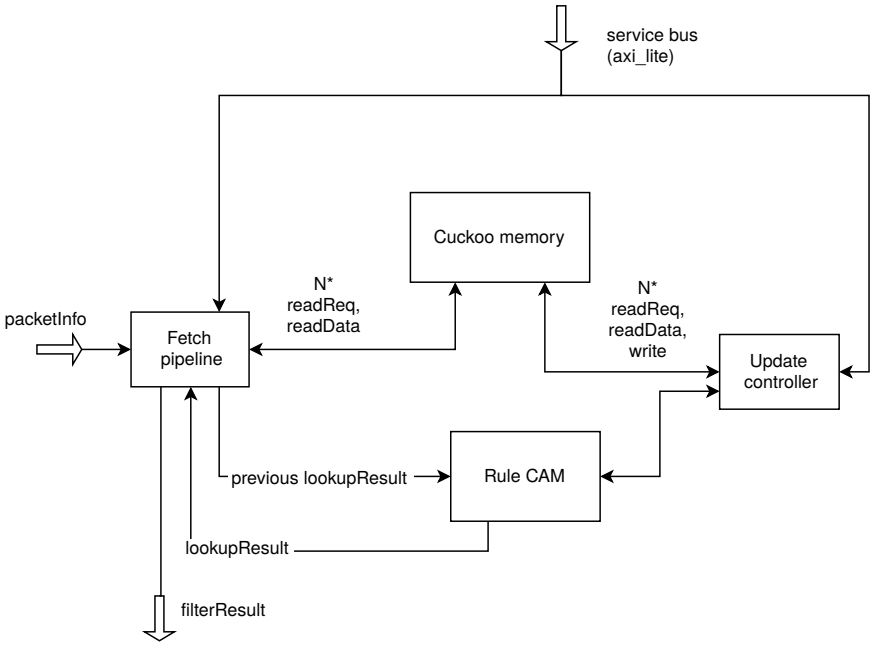


Obrázek 6.14: Struktura vyhledávací jednotky pro zřetězenou implementace L4 filtru.

Tyto údaje platí jen při vyhledávání, vkládání a mazání musí počkat na vyprázdnění celé linky a až pak může vykonat svou akci, jinak by mohlo dojít k nekonzistenci obsahu tabulky. Kvůli těmto nevýhodným vlastnostem filtr nebyl dokončen. V aplikacích, které nevyžadují časté vkládání, by tato implementace byla efektivnější.

6.7 Záznam filtru

Záznam filtru je položka uložená v paměti ve vyhledávacích tabulkách filtru. Je složená z klíče, jednobitového příznaku značícího platnost a pravidla.



Obrázek 6.15: Struktura zřetězené implementace L4 filtru.

6.7.1 Klíč

N-tice $(Source, Destination)$, kde obě položky jsou odvozeny z třídy rozhraní `Endpoint`. Tato rozhraní je n-tice $(IP, PORT, IP\_IS\_V6)$[5]:

- `IP` – IPv4 nebo IPv6 adresa v obou případech je použito 128 bitů, little-endian zarovnaní v případě IPv4, avšak vlastní hodnota je zarovnaná tak, jak je zarovnána na Ethernetu, tedy big-endian.
- `PORT` – port 16b.
- `IP_IS_V6` - příznak indikující, že v IP je uložená IPv6 adresa.

6.7.2 Pravidlo

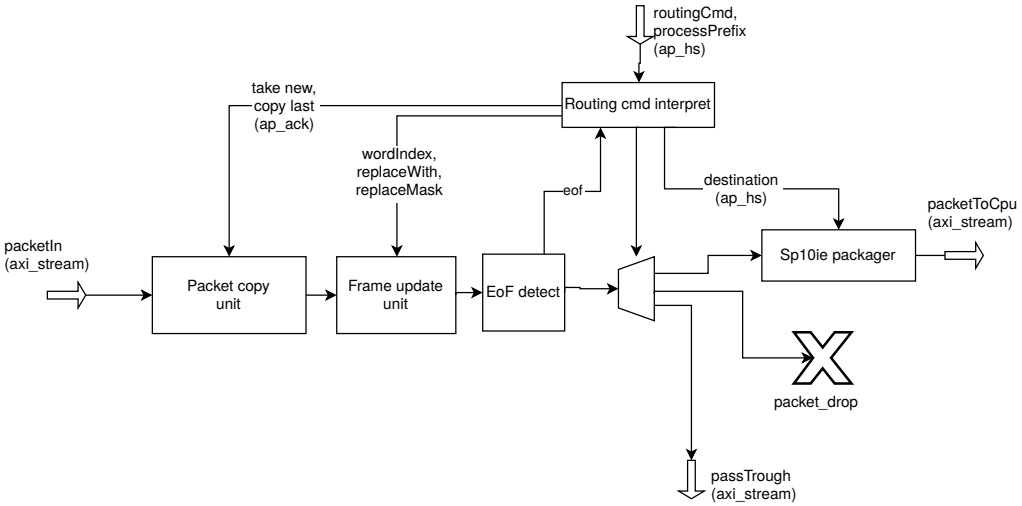
Pravidlo z filtru reprezentuje akci, kterou má filtr provést s daným paketem. Je specifikováno následujícím formátem:

Příkaz (bit 0)	Argument (zbytek)
`ROUTE` =0	`PROCESS_BITMASK`
`PROCESS`=1	`DROP`=0, `PASS`=1

Šířka `PROCESS_BITMASK` tedy i šířka celého pravidla je volitelná a konfigurovatelná na úrovni kódu pro FPGA. Tedy změnou `FilterRule.ARG_WIDTH` se dá nastavit maximální počet připojitelných výpočetních jednotek.

V rámci této práce byly implementovány následující kombinace příkazů:

[5]Položky jsou uskupeny tak, aby bylo možné zajistit podporu symetrického hašování a zároveň bylo možné lehce použít porovnávací operátory.



Obrázek 6.16: Struktura jednotky Packet internal router.

- `PROCESS` - Argumnent `DROP` zahodí paket. `PASS` propustí paket beze změny rovnou na vnější síť.
- `ROUTE` - podle argumentu `PROCESS_BITMASK` je paket odeslán na všechny procesy jejichž bit je v masce nastaven na log. 1.

6.8 Packet internal router

Tato jednotka provádí všechny routovací příkazy. Její struktura je znázorněna na obrázku 6.16. Příkazy pro tuto jednotky jsou generovány na základě výsledků filtrovacího jádra a registrovaných worker procesů. Dokáže kopírovat daný ethernetový rámec na více procesorových jednotek a dokáže provádět jednoduché úpravy v daném rámci, tato možnost prozatím není plně využita. Pro datové streamy jsou použity Axi4-stream rozhraní. Rozhraní pro příkaz je v kodu popsáno třídou `RoutingCmdIntf`. Jedná se o rozhraní, které využívá handshake synchronizaci, obsahuje následující datové signály:

- `takeNew` – Signalizuje, jestli se má z bufferu načíst a odeslat nový rámec, nebo jestli se má odeslat předchozí.
- `cmd` – Vlastní příkaz, implementovány jsou `ROUTE` (přeposlání na výpočetní jednotku), `DROP` (zahození rámce), `PASS` (přeposlání na opačný ethernetový port).
- `targetMac` – Pro příkaz `ROUTE` nese adresu cílového výpočetního modulu.

6.9 Spotřebované zdroje FPGA

Následující údaje jsou uvedeny bez bufferů. V architektuře je jen jeden buffer, který je nutný (uvnitř paketového procesoru před jednotkou Packet internal router). Tento buffer zabírá v minimální konfiguraci 1 Block RAM, 27 FF a 51 LUT. Tabulka 6.9 obsahuje výsledky syntézy akcelerátoru, jak je zakresleno ve schématu 6.2. Tabulka 6.9 obsahuje výsledky syntézy akcelerátoru a periferií, jak zobrazuje obrázek 6.1.

Zdroj	Využito
FF	3278
LUT	4568
Memory LUT	1568
BRAM	35

Tabulka 6.1: Spotřebované zdroje pro akcelerátor v kofiguraci: Výchozí filtr, paralelní L4 filtr se čtyřmi tabulkami po 1024 položkách v BRAM, 16 položková CAM.

Zdroj	Využito
FF	26508
LUT	26194
Memory LUT	5768
BRAM	35
BUFG	5
MMCM	1
PLL	1
GT	4

Tabulka 6.2: Spotřebované zdroje pro kompletní design (konfigurace stejná jako v případě tabulky 6.9).

Kapitola 7

Modifikace akcelerátoru

7.1 Legální odposlechy

Legální odposlechy většinou sledují konkrétní adresu, protokol nebo vzor v datových částech paketů [4]. Pro detekci těchto vzorů je potřeba rozšířit stávající řešení o další jednotku. Její výstup reprezentující nalezený regulární výraz musí být přiveden do L4 filtru, ve kterém se musí klíč rozšířit o výsledek této jednotky.

Legální odposlechy vyžadují dynamickou tvorbu velkého množství pravidel, proto je vhodné použít L4 filtr s pamětí uloženou v QDR. Prakticky se tedy jedná o připojení jednotky pro vyhledávání výrazů, přepnutí L4 filtru do QDR módu a rozšířením třídy *FilterKey* o další položku. Zbytek může zůstat nezměněn.

7.2 Firewall

Pro paketový filtr není potřeba udělat žádnou modifikaci. Případně je možné nahradit L4 filtr filtrem na bázi adresových rozsahů [6], který může být pro tuto aplikaci efektivnější. Pro stavový firewall by bylo potřeba doplnit protokol paketu do klíče filtru. Tím začne filtr rozlišovat protokoly.

Pro aplikační firewall je nutno doplnit jednotku pro vyhledávání vzorů jako v případně legálních odposlechů 7.1.

7.3 Distributor zátěže

Cílem je rovnoměrně rozložit datové toky mezi výpočetní jednotky. Na tuto aplikaci není potřeba L4 filtr. Výchozí filtr stačí přepnout do režimu hašování. To bude mít za následek to způsobí rozdělení paketů podle haše (CRC16) hlavičky paketu.

7.4 NAT plně realizovatelný v FPGA

Tato úprava vyžaduje rozšíření o několik pokročilých komponent, které se snadno připojí do stávajícího systému.

- Dodat do L4 filtru 6.3 jednotku pro generování časových značek.
- Upravit kontrolér L4 fitru tak, aby při nenalezení pravidla automaticky přidal nové a k němu i obrácené pravidlo. A při nalezení pravidla se aktualizuje časová značka.

- Rozšířit záznam ve filtru (viz 6.7.2) o položku časové značky.
- Rozšířit výsledek filtru a příkaz pro routování o příkaz aktualizace zdrojové/cílové IP, portu.
- Doimplementovat do jednotky *Routig cmd interpret* (viz 6.8) výše zmiňované příkazy.

Záznam ve filtru bude také nutné rozšířit o příznak, kterým se odliší automaticky přidávané pravidla od statických.

Pravidla ve filtru nebude možné odmazávat po vypršení timeout, protože tento stav nelze detekovat. Místo pro záznamy v tabulkách tedy nelze uvolňovat. Při kolizi v tabulce lze však zkontrolovat, jestli časová značka kolidujícího záznamu nepropadla na timeout. Pokud ano, k záznamu lze dohledat opačný a ten taktéž smazat. V případě, že časová značka bude aktuální, aplikuje se standardní vkládací algoritmus tak, jak je popsán v L4 filtru s přídavnou detekcí timeoutu.

Existuje více možností, jak rozdělovat dostupný adresový prostor pro nové pravidla NAT. Odlišné případy nasazení mohou vyžadovat jiný přístup. Nejjednodušší možností je použít čítače a v průběhu přiřazování iterovat přes celý adresový prostor. Tento přístup je výhodný zejména v případech, kde nelze zkontrolovat, zda jsou adresy právě používány jiným pravidlem a současně je smazání právě používaného pravidla přijatelné. Výhoda tohoto řešení spočívá v jeho nízké náročnosti na zdroje FPGA.

V případě, že je počet některých adres dostatečně malý na to, aby se dal uložit do bitmapy lze použít specializované jednotky pro detekci první jedničky pro vybrání adresy a pod.

V případě, že je nutné kontrolovat použité adresy a zároveň není možné je zkontrolovat dříve zmiňovaným přístupem, lze si jednoduše vytvořit další tabulky používající kukaččí haš, ve kterých bude uložena vždy konkrétní adresa. Při vytváření pravidla by se iterovalo čítačem přes dostupné adresy a první, která by neměla záznam v tabulkách použitých adres, by byla použita a do těchto tabulek vložena. Při mazání pravidla z filtru se tedy bude muset odstranit záznam i z těchto tabulek. L4 filtr je vysoce flexibilní a přepsáním tříd rozhraní *FilterKey* a *FilterRule* a jejich aktualizací v instanci filtru stačí na to, aby byly naimplementovány vyhledávací tabulky zmíněné výše.

Poslední řešení je preferované. Pravidlo filtru se zvětší o 121 b (v případě IPv6). Současně může být aktivní velké množství pravidel. Je tedy vhodné použít QDR.

Kapitola 8

Závěr

Vzniklý akcelerátor dokáže filtrovat provoz na úrovni rychlosti linky. NAT je akcelerován filtrací nežádoucích paketů a rozdělováním paketů na jednotlivé výpočetní moduly. Samotná aktualizace adres je ponechána na výpočetních jednotkách.

Pro platformu poslední verze je implementován model na úrovni všech datových toků v celém akcelerátoru. Tento model primárně slouží k vysokoúrovňovému návrhu zařízení a k testování funkčnosti hardware. Na tomto modelu probíhaly statistické testy a testování optimalizací.

Implementace paměti a hašovací funkce jsou naimportovány z VHDL. Zbytek implementace je generován pomocí skriptů.

V rámci této práce byl navržen design celého systému i designu akcelerátoru, který byl následně implementován. Implementován je celý design pro filtrování i NAT, to znamená paketový procesor i filtrovací jádro. Jedinou vyjímku tvoří komponenta HFE, která je tvořena zvlášť. Přehled hlavních komponent je v kapitole 6. Dále je implementována redukce z rozhraní z USB čipu na Axi4-Lite. 10G ethernet MAC a PCS/PMA, QDR řadič a Axi4 interconnect jsou již dostupné od Xilinx. CAM a hašovací funkce byly převzaty z Sprobe. Sestaveny byl top-level design se čtyřmi ethernetovými porty, USB a QDR. Deska akcelerátoru není vyrobena, ale pro její otestování je již připraven testovací design, kterým bude možné otestovat až 8 SPI+ portů, QDR a komunikaci přes USB.

Řada jednotek má jen triviální testy a tedy nejsou plně otestovány a zverifikovány. Na druhou stranu je naprogramován simulátor chování celého systému (a tedy i jednotlivých komponent). Tyto simulace simulují chování kódu, ze kterého probíhá generování VHDL. Byly použity 4 simulátory. Pro simulaci VHDL bylo použito Vivado. Pro simulaci v Pythonu byly použity MyHdl, simulátor datových streamů a netlistů z Hw toolkit. Koncept je tedy možné částečně testovat pomocí unittestů v Pythonu. Experimentování se zařízením se tím velmi výrazně zjednodušilo oproti VHDL/Verilogu i SystemVerilogu. Akcelerátor je schopen pracovat na 156,25 MHz při datové šířce 64 bitů, tedy lze jej provozovat na 10G sítích.

Díky skriptům v Pythonu je možné kteroukoli jednotku vyexportovat jako konfigurovatelné IP jádro a následně ji použít v jiném projektu.

Kapitola 6.9 obsahuje nároky filtrovacího jádra na zdroje FPGA.

Literatura

[1] Consortium, Q.: QDR Technology [online]. http://www.qdrconsortium.org/qdr-technology.html, 2015-11-30 [cit. 2016-1-2].

[2] Cypress Semiconductor Corporation: *CY7C1526KV18 CY7C1513KV18 CY7C1515KV18 72-Mbit QDR® II SRAM Four-Word Burst Architecture [online].* 2016-01-19, http://www.cypress.com/file/39896/download.

[3] K. Girish, B. V., M. John: QDR-II, QDR-II+, DDR-II, and DDR-II+ Design Guide [online]. http://ee-classes.usc.edu/ee454/SSRAM/Cypress_ISSI_SSRAMs/Cypress/AN4065_001-15486.pdf, 2015-11-30 [cit. 2016-1-2].

[4] Karpagavinayagam, B.; State, R.; Festor, O.: Monitoring Architecture for Lawful Interception in VoIP Networks. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, Červenec 2007, s. 5–5, doi:10.1109/ICIMP.2007.27.

[5] Kekely, L.; Kučera, J.; Puš, V.; aj.: Software Defined Monitoring of Application Protocols. *IEEE Transactions on Computers*, ročník 65, č. 2, Únor 2016: s. 615–626, ISSN 0018-9340, doi:10.1109/TC.2015.2423668.

[6] Lampson, B.: IP Lookups using Multiway and Multicolumn Search. *ACM Transactions on Networking*, ročník 7, č. 3, Červen 1999: s. 324–334, http://research.microsoft.com/en-us/um/people/blampson/61-IPLookup/61-IPLookup.pdf.

[7] Martinez, C. J.; Lin, W. M.; Patel, P.: Optimal XOR hashing for a linearly distributed address lookup in computer networks. In *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, Říjen 2005, s. 203–210, doi:10.1145/1095890.1095919.

[8] P. Srisuresh, K. E.: The IP Network Address Translator (NAT). RFC 3022, The Internet Engineering Task Force (*IETF®*), Leden 2001. URL http://tools.ietf.org/html/rfc3022

[9] Pagh, R.; Rodler, F. F.: Cuckoo hashing. *Journal of Algorithms*, ročník 51, č. 2, 2004: s. 122 – 144, ISSN 0196-6774, doi:http://dx.doi.org/10.1016/j.jalgor.2003.12.002. URL http://www.sciencedirect.com/science/article/pii/S0196677403001925

[10] Pontarelli, S.; Reviriego, P.; Maestro, J. A.: Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput. *IEEE Transactions on Computers*, ročník 65, č. 1, Leden 2016: s. 326–331, ISSN 0018-9340, doi:10.1109/TC.2015.2417524.

[11] Singh, M.; Garg, D.: Choosing Best Hashing Strategies and Hash Functions. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, Březen 2009, s. 50–55, doi:10.1109/IADCC.2009.4808979.

[12] Thinh, T. N.; Kittitornkun, S.; Tomiyama, S.: Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, Prosinec 2007, s. 121–128, doi:10.1109/FPT.2007.4439240.

[13] Wu, Q.-X.: The Research and Application of Firewall based on Netfilter. *Physics Procedia*, ročník 25, 2012: s. 1231 – 1235, ISSN 1875-3892, doi:http://dx.doi.org/10.1016/j.phpro.2012.03.225, international Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao.
URL http://www.sciencedirect.com/science/article/pii/S1875389212006414

[14] Xilinx Inc.: *7 Series CLB*. 2014-11-17, http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[15] Xilinx Inc.: *7 Series Transceivers*. 2015-08-19, http://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf.

[16] Xilinx Inc.: *10G Ethernet MAC v15.1*. 2016-04-06, http://www.xilinx.com/support/documentation/ip_documentation/ten_gig_eth_mac/v15_1/pg072-ten-gig-eth-mac.pdf.

[17] ŠTĚPÁN, J.: Měření rychlosti internetu v ČR [online]. http://www.vse.cz/vskp/eid/48343, 2015-11-30 [cit. 2016-1-2].

Přílohy

Seznam příloh

Příloha A

Obsah DVD

- `designs/` - obsahuje Vivado (nástroj použitý při vývoji pro FPGA) projekty testovacích a hlavních designů pro celé zařízení.
- `doc/` - obsahuje text této práce
- `src/hdl` - obsahuje kod převážně ve VHDL, který vznikl v rámci této práce
- `src/hls` - obsahuje skripty, které slouží ke generování HDL a testy

Příloha B

Framework Hw toolkit

Hw toolkit je framework napsaný v Pythonu 3 usnadňuje vývoj systémů používající FPGA. Primárně slouží pro práci s netlisty na vysoké úrovní abstrakce. Do interní reprezentace netlistů Hw toolkitu lze importovat kód popsaný jazyky VHDL a Verilog. Zároveň je možné z netlistů zpětně generovat syntetizovatelné VHDL. Vývoj tohoto frameworku je spjat s vývojem akcelerátoru rámci této práce. Obsahuje řadu utilit, které lze použít k vykonávání rutinních činností spojených s návrhem a implementací designu. Použitá je např. automatická propagace resetovacích a hodinových signálů, generování registrů na sběrnicích, pole rozhraní, automatická propagace generik a podobně.

Tento framework je založen na rozhraních (Interface) a jednotkách (Unit). Rozhraní mohou být libovolně zanořována a tvoří kontejnery pro signály a generika. Tyto třídy v sobě nesou i definice směrů jednotlivých signálů z čehož lze odvodit směr celého rozhraní. Nad objekty rozhraní jsou implementovány např. operace připojení, zabalení a rozbalení. Jednotky ohraničují části netlistu. Tyto objety lze v Pythonu vytvořit nebo je možné je naimportovat z VHDL, Verilog, MyHDL[1] nebo z Vivado HLS. Jednotky je možné konvertovat do HDL jazyků pomocí implementovaných serializerů např. pomocí funkce `toRtl`.

Framework obsahuje širokou škálu přeimplementovaných rozhraní a příkladů a testů. Některé komponenty z příkladů jsou použity v implementaci této práce (FIFO, Dual port BRAM, Single port BRAM, generátor AXI4-lite registrů).

Simulační část (úroveň streamů i úroveň signálů) framworku je taktéž užita v této práci.

Ukázka definice rozhraní a jednotky je uvedena v kodu B.1.

[1] nástroj pro návrh hadrware v Pythonu http://www.myhdl.org/

```
class MyInterface(Interface):
    def _config(self):
        #optional, initialize configurable parameters
        pass

    def _declr(self):
        # declare all interface objects which are part of this interface
        self.rx = Signal(masterDir=DIRECTION.IN)
        self.tx = Signal()


class MyUnit(Unit):
    def _config(self):
        #optional, initialize configurable parameters
        pass

    def _declr(self):
        # declare all subunits and interfaces,
        # which should be visible for parent
        self.dataIn = MyInterface(isExtern=True)
        self.dataOut = MyInterface(isExtern=True)

    def _impl(self):
        # signals rx and tx are connected by interface specification
        connect(self.dataIn, self.dataOut)
```

Algoritmus B.1: Ukázková definice rozhraní a jednotky v `Hw toolkit`.