VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

STAVOVÉ ZPRACOVÁNÍ PAKETŮ V JAZYCE P4

THE STATEFUL PACKET PROCESSING IN P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVEL KOHOUT**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kohout Pavel**
Program: Informační technologie
Název: **Stavové zpracování paketů v jazyce P4**
The Stateful Packet Processing in P4 Language
Kategorie: Návrh číslicových systémů

Zadání:

1. Seznamte se s jazykem P4, jeho překladačem do firmware pro FPGA a konfigurační knihovnou libp4dev.
2. Navrhněte rozšíření překladače o podporu registrů a čítačů.
3. Implementujte navržené rozšíření překladače.
4. Rozšiřte konfigurační knihovnu o funkce pro zápis/čtení hodnot registrů a čítačů.
5. Implementovaná rozšíření otestuje na akcelerační síťové kartě s FPGA čipem.
6. Diskutujte dosažené výsledky a možnosti jejich dalšího rozšíření.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kořenek Jan, doc. Ing., Ph.D.**
Konzultant: Benáček Pavel, Ing., CESNET
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 26. října 2018

Abstrakt

S rostoucími rychlostmi a komplexností počítačových sítí vznikají požadavky na vytváření výkonných zařízení, která jsou schopna provádět sběr statistik a měnit svoji funkcionalitu podle požadavků síťových administrátorů. Tyto požadavky mohou být popsány pomocí specializovaných programovacích jazyků, například jazykem P4. V rámci této bakalářské práce byl proveden návrh, implementace, testování a integrace modulů stavových pamětí registrů a čítačů do systému překladače jazyka P4 do technologie FPGA. Vytvořený systém podporuje sběr statistik popsaných v jazyce P4 na rychlosti až 100 Gb/s.

Abstract

With the growing speed and complexity of computer networks, arise requirements for creating powerful devices that are capable of collecting statistics and changing their own functionality according to the demands of network administrators. These requirements can be described using specialized programming languages such as P4. In this bachelor thesis a design, implementation, testing and integration of register and counter stateful memory modules into P4 compiler system for FPGA technology was made. The created system supports the collection of statistics described in P4 language at speeds up to 100 Gbps.

Klíčová slova

P4, stavové zpracování, Liberouter, FPGA

Keywords

P4, stateful processing, Liberouter, FPGA

Citace

KOHOUT, Pavel. *Stavové zpracování paketů v jazyce P4*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jan Kořenek, Ph.D.

Stavové zpracování paketů v jazyce P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Jana Kořenka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Kohout
14. května 2019

Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce Doc. Ing. Janu Kořenkovi, Ph.D. za jeho pomoc a podporu při psaní této práce. Dále bych chtěl poděkovat kolegům z výzkumného týmu Liberouter. Velké poděkování patří Ing. Pavlu Benáčkovi, Ph.D. za jeho odborné rady a pomoc při návrhu a integraci modulů stavových pamětí.

Obsah

Kapitola 1

Úvod

V dnešní době roste důraz na rychlost zpracování síťového provozu a s ní spojená efektivnost správy síťové infrastruktury. S tímto trendem se současně zvyšuje počet aplikací, které si dávají za cíl rekonfiguraci síťové infrastruktury pro zvýšení její bezpečnosti, usnadnění změn směrování datových toků a vylepšení jejich další parametrů. Rekonfigurace síťové infrastruktury je problematická činnost, jelikož chování síťové infrastruktury je definováno jednotlivými autonomními síťovými zařízeními. Síťová zařízení jsou vzájemně provázaná a při požadavku na změnu chování celé sítě je zapotřebí změnit konfiguraci často více zařízení. Provázanost zařízení klade velké nároky na administrátory síťové infrastruktury, kteří se musí velmi detailně orientovat v celé síti a znát provázání jednotlivých komponent, aby byli schopni měnit chování sítě bez porušení stávající funkčnosti. Popsaný problém pomáhá řešit architektura Software-defined networking (SDN).

Jednou z možností, jak můžeme realizovat architekturu SDN, je použití vysokoúrovňových doménově specifických programovací jazyků, které slouží pro definici funkcionality síťových zařízení. Tyto jazyky musí nabízet širokou nabídku funkcí pro pokrytí co nejvíce uživatelských požadavků na funkcionalitu dané sítě. Mezi ty v současnosti nejžádanější a nejvíce preferované funkcionality patří vysoká výkonnost a bezpečnost počítačových sítí. Moderní páteřní sítě dosahují rychlostí stovek gigabit, což klade velké nároky na technické a programové vybavení sítě. Univerzální procesory nedokáží při vysokých rychlostech efektivně plnit jejich úlohu, a proto je nutné procesory nahradit jinými vhodnějšími zařízeními, které dokáží řešit dané úkony efektivně i na vysokých rychlostech díky schopnosti vysoké paralelizace. Ideální platformou pro reflexi požadavků na výkonnost a flexibilnost síťové infrastruktury je programovatelné hradlové pole FPGA. Proto by pro mapování výše zmíněných jazyků měly vznikat překladače, které jsou schopné jejich abstraktní popis namapovat na tuto platformu.

Bezpečnost v sítích může být založena na identifikaci a klasifikaci pouze jednotlivých datových rámců - paketů. To je v dnešní době nedostatečné, jelikož přístup omezení se pouze na jednotlivé pakety může být využit útočníkem, kterému stačí rozložit vzorky charakteristické pro útok do více paketů. Řešením popsaného problému je přístup, kdy se na datový tok díváme v kontextu jeho celého, díky čemuž slabinu předchozího přístupu eliminujeme. Stavové zpracování, jak se druhý zmíněný přístup nazývá, je další důležitou vlastností, kterou by měly výše zmíněné jazyky splňovat.

Jedním z jazyků, které splňují výše zmíněné požadavky je jazyk P4, za jehož vznikem stojí organizace *P4 Language Consortium* [19]. Jazyk P4 je platformě nezávislý a slouží k popisu zpracování síťového provozu síťovými prvky. K jazyku byl vytvořen i lexikální a syntaktický analyzátor, který je veřejně dostupný na webových stránkách organizace.

Analyzátor vytváří objektovou reprezentaci funkcionality popsaného síťového zařízení, která musí být následně namapována na cílovou technologii.

Cílem této práce je rozšíření podporované funkcionality překladače jazyka P4, vyvíjeného v organizaci CESNET pro platformu FPGA, o funkce týkající se stavového zpracování pro propustnost zařízení až 100 Gb/s. Pro tento účel byly navrhnuty, implementovány a otestovány komponenty registrů a čítačů realizující stavové prvky v jazyce P4 a další pomocné komponenty umožňující začlenění do již existující architektury. Nad rámec zadání byla navržena a implementována stavová paměť měřičů.

Tato práce byla rozčleněna do sedmi kapitol, které mají za cíl poskytnout informace potřebné k pochopení kontextu a realizaci celého díla. V kapitole 2 je blíže popsána motivace vzniku jazyka P4 s jeho podrobným popisem. Kapitola dále blíže popisuje využité technologie, které bylo potřebné nastudovat. Na základě znalostí technologií a vlastností jazyka P4 byl v kapitole 3 sestaven soupis požadavků na návrh a začlenění stavových pamětí do systému. V kapitole 4 je uveden detailní popis návrhu jednotlivých komponent stavových pamětí včetně pomocných komponent umožňující korektní začlenění do existujícího systému jazyka P4. Celková integrace je blíže popsána v kapitole 5. Kapitola 6 obsahuje popis verifikace a dosažených výsledků při výkonnostním testování. V závěru práce v kapitole 7 jsou shrnuty výsledky a navrhnut další možný postup při vývoji stavových pamětí v jazyce P4.

Kapitola 2

Seznámení s problematikou

Správa síťové infrastruktury je komplexní záležitost. Pro jejich efektivní realizaci jsou zaváděny důmyslné řešení sjednocující přístup k všem prvkům infrastruktury. Příkladem takového řešení je *Software–defined networking* s využitím konfiguračního protokolu *OpenFlow*. Specializované jazyky, mezi které patří i jazyk P4, jsou vhodným nástrojem pro konfiguraci zařízení pomocí protokolu *OpenFlow*. Poskytují totiž možnost upravit obecně zapsanou funkcionalitu na efektivní implementaci pro konkrétní zařízení. Umožňují tak jednotný přístup k prvkům sítě současně s maximálním využitím jejich potenciálu.

2.1 Software–defined networking

SDN architektura poukazuje na problémy architektury tradičních sítí, které jsou decentralizované a komplexní, i přes zvyšují se požadavky na jejich efektivní a jednoduchou správu. SDN nabízí řešení v rozdělení provozu do řídící a datové vrstvy, kdy řídící vrstva centralizuje celou logiku sítě(viz obrázek 2.1). To přináší následující výhody:

Přímá konfigurovatelnost - oddělení řídící a datové vrstvy

Abstrakce - přináší výhodu v konfiguraci, protože není nutné znát přesnou topologii sítě a konfigurace je snadná díky komunikaci kontrolní a datové vrstvy přes společné API.

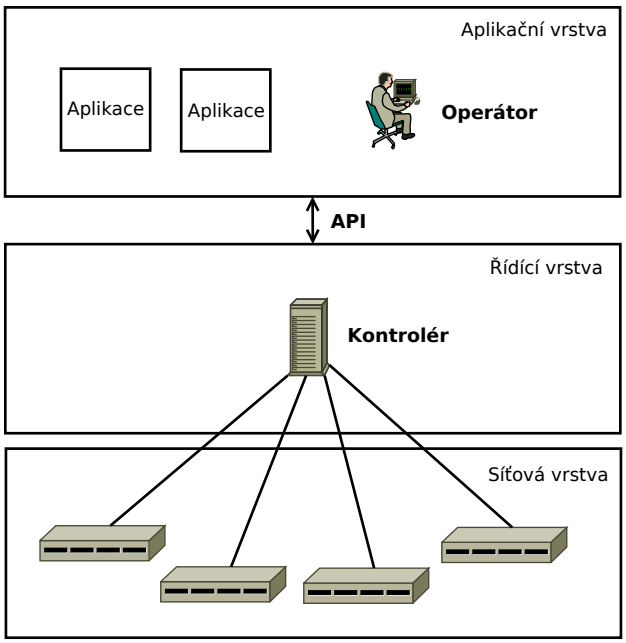
Centralizace - lepší správa sítě a toku dat díky centralizované logice v řídící vrstvě

Otevřený standard - Nezávislé na výrobci síťového hardware

2.1.1 OpenFlow protokol

Komunikace mezi řídící a datovou vrstvou probíhá za využití OpenFlow protokolu(aktuálně ve verzi 1.5.1 [14]), pomocí kterého dochází ke konfiguraci síťových prvků - přepínačů. To se děje na základě vkládání pravidel do tabulek, podle kterých přepínače směrují každý datový tok. Každá verze protokolu OpenFlow definuje podporované protokoly a akce. Přehled jednotlivých verzí protokolu a jejich podporovaných protokolů je uveden v tabulce 2.1.

Hlavní nevýhodou OpenFlow protokolu je uzavřenost množiny podporovaných protokolů a akcí. Pro přidání nového podporovaného protokolu či akce je nutné aktualizovat standard, což může být poměrně dlouhý proces. Rozšíření sady podporovaných protokolů a akcí je snadno proveditelné v případě softwarové implementace přepínače. V případě hardwarové implementace uplatňované ve vysokorychlostních sítích je situace podstatně složitější



Obrázek 2.1: Architektura SDN [20].

Verze	Datum	Podporované protokoly
1.0	Prosinec 2009	12 políček protokolů (Ethernet, IPv4, ...)
1.1	Únor 2011	15 políček protokolů (přidáno MPLS, metadata mezi tabulkami)
1.2	Prosinec 2011	36 políček protokolů (ARM, ICMP, IPv6,. . .)
1.3	Červen 2012	40 políček protokolů)
1.4	Říjen 2013	41 políček protokolů
1.5	Prosinec 2014	45 políček protokolů

Tabulka 2.1: Verze Openflow protokolu [20]

a velmi často může vést na výměnu starého zařízení za novější. Právě tento problém pomáhá řešit jazyk **P4**.

2.2 Jazyk P4

Jazyk P4 (**P**rogramming **P**rotocol-independent **P**acket **P**rocessors) je vysokoúrovňový, doménově specifický jazyk, sloužící pro programování protokolově nezávislých síťových procesorů. Mezi hlavní výhody tohoto jazyka patří schopnost volně definovat zpracování paketů síťových zařízení. Dalšími neopomenutelnými výhodami je nejen výše zmíněná programovatelnost síťových zařízení, ale také protokolová a platformní nezávislost. V době psaní této práce existovaly dva standardy jazyka P4:

$P4_{14}$ Tento standard je první verzí jazyka P4 a byl vystavěn kolem abstraktního modelu. $P4_{14}$ definuje sadu primitivních akcí, které není možno už dále rozšiřovat. Celý kon-

cept $P4_{14}$ je poměrně flexibilní, ale v některých případech stále příliš statický kvůli chybějící možnosti rozšíření o nové primitivní akce.

$P4_{16}$ Novější standard jazyka P4 se snaží řešit nedostatky starší verze $P4_{14}$ tím, že nezavádí žádný abstraktní model a nechává uživateli možnost si definovat zapojení jednotlivých částí zařízení. Navíc nabízí možnost si volně definovat vlastní funkcionalitu pomocí jazykové konstrukce *extern*.

2.2.1 Platformní nezávislost

Platformní nezávislost implementace standardů jazyka P4 umožňuje mapovat popsanou funkcionalitu na více typů zařízení jako jsou CPU, NPU a FPGA (bude vysvětleno později). Uvedená zařízení jsou označována jako cílová (P4 targets) a každé musí být poskytnuto spolu s kompilátorem, který mapuje zdrojový kód jazyka P4 na zdroje konkrétního zařízení.

Na stránkách komunity kolem jazyka P4 jsou poskytovány základní nástroje, které mohou být použity v jiných projektech při tvorbě kompilátorů. Jedním z nástrojů je i knihovna P4-HLIR [16], která vytváří model pomocí objektů v jazyce Python. Tato knihovna obsahuje lexikální a syntaktický analyzátor jazyka P4 a tím usnadňuje vývoj překladače pro cílovou platformu.

CPU

Nejčastěji cílenými zařízeními pro vykonávání P4 kódu jsou Central Processing Units(CPUs). CPU je vybíráno jako cílové zařízení díky jeho flexibilitě a jednoduchosti nasazení s minimálním úsilím. Navíc CPU je možno sdružovat do tzv. clusterů a tím zvyšovat celkovou výkonnost výpočetního systému. Nevýhodou popsaného přístupu je špatná škálovatelnost pro zpracování síťových dat v reálném čase. Projekt P4C-BEHAVIORAL [17] je ukázkou implementace překladače pro CPU.

NPU

NPU (Network Processing Unit) je další velmi časté cílové zařízení pro P4. NPU je zařízení uzpůsobené pro nasazení v síťové doméně díky rychlé a efektivní implementaci vyhledávání vzorů, bitových posunů a dalších operací. Jednotky NPU jsou schopny zpracovávat síťovou komunikaci na rychlostí až 100 Gbps. Implementací překladače pro NPU poskytuje například společnost Netronome [13].

FPGA

FPGA (Field-Programmable gate array) je voleno jako cílené zařízení při potřebě vysokého výkonu a flexibilitě zpracování. Programovatelnost a výkonnost FPGA z něj dělá vhodný nástroj v oblasti počítačových sítí, protože pomocí HDL jazyka (Hardware Description Language) můžeme popsat požadovanou funkcionalitu zařízení. HDL popis lze vygenerovat pomocí více abstraktnějšího popisu jazyka P4. Konstituce samotného FPGA umožňuje dosahovat výkonnosti zpracování až 100 Gb/s a více, což z něj dělá vhodnou platformu pro výzkum a akceleraci zpracování dat v oblasti vysokorychlostních sítí. Příkladem kompilátoru pro FPGA obvody je projekt P4FPGA, který je poskytován komunitou kolem jazyka P4 a je dostupný na [18]. Implementací kompilátoru $P4_{14}$ pro FPGA se zabývá i výzkumná skupina Liberouter [11] sdružení CESNET [6]. Vyvíjený překladač je popsán v disertační práci Pavla Benáčka [4].

2.2.2 $P4_{14}$

Standard $P4_{14}$ je založen na abstraktním modelu, který je uveden na obrázku 2.3. Abstraktní model je složen z parseru, deparseru a posloupnosti match+action tabulek, umístěných mezi vstupem a výstupem. Úkolem parseru je identifikovat hlavičky u příchozího paketu a připravit je pro budoucí zpracování v posloupnosti match+action tabulek. Každá match+action tabulka vyhledává v námi zvolené podmnožině hlaviček zpracovávaného toku a provede akci odpovídající záznamu tabulky.

Každé síťové zařízení lze v jazyce $P4_{14}$ definovat pomocí pěti základních aspektů tohoto zařízení:

Definice struktury hlaviček protokolů se kterými bude dané síťové zařízení pracovat.

Specifikace procesu parsování vyjádřeného jako konečný automat, u kterého přechody mezi jednotlivými uzly jsou podmíněny námi zvolenými políčky právě zpracovávaného protokolu. Koncový stav automatu spustí tzv. kontrolní program.

Specifikace tabulky s pravidly pro výběr akce, který se má na daný síťový provoz aplikovat. Nejdříve se příchozí data převedou na klíč, podle kterého se v tabulce provádí hledání nejvhodnějšího pravidla. Klíč samotný je v jazyce rozložen na části, u kterých se specifikuje způsob porovnávání.

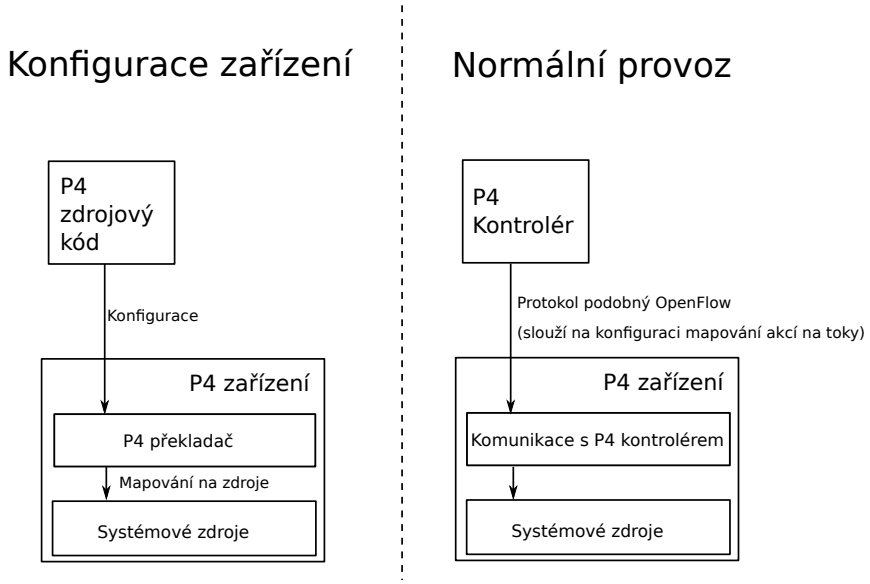
Specifikace akcí prováděných nad tokem, který byl na tuto akci namapován. Standard $P4_{14}$ nám nabízí sadu základních tzv. primitivních akcí, které poskytují dostatečnou funkcionalitu pro práci s datovými toky. Dále uživatel může definovat vlastní akce, které se mohou skládat z více primitivních akcí a dalších již definovaných akcí uživatele.

Specifikace kontrolního programu propojuje vše předešlé dohromady a tvoří vstupní blok programu, který určuje pořadí, ve kterém budou jednotlivé tabulky aplikovány. Pořadí může být ovlivněno splněním/nesplněním podmínky, nalezením/nenalezením záznamu v tabulce, atd.

Nasazení jazyka $P4_{14}$ na zařízení je rozděleno do dvou fází, kterými jsou konfigurace síťového zařízení a normální provoz. Na levé části obrázku 2.2 je zachycena fáze konfigurace, během které se převádí program napsaný v $P4_{14}$ do interní reprezentace, co nejvhodnější pro dané zařízení. Poté přechází zařízení do fáze normálního provozu zachyceného na obrázku 2.2 vpravo, ve které kontrolér řídí činnost zařízení prostřednictvím plnění tabulek zadanými pravidly.

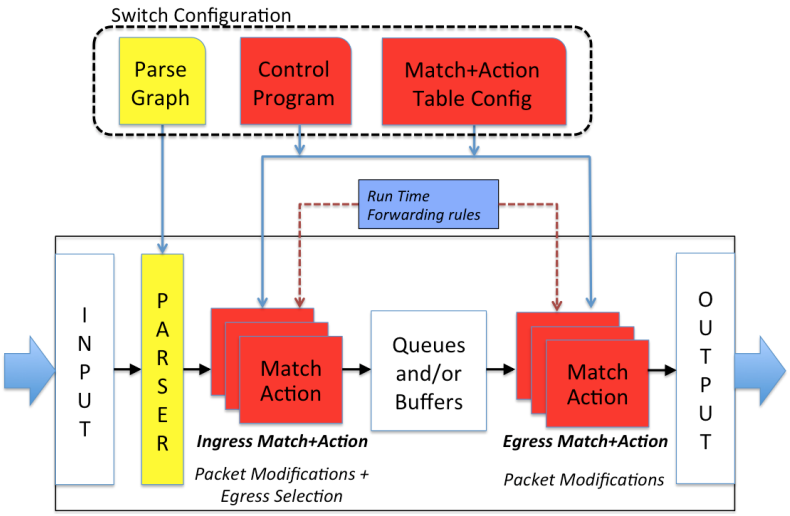
Abstraktní model

Abstraktní model $P4_{14}$ zařízení je znázorněn na obrázku 2.3. Obsahuje parser, deparser, match+action tabulky a mechanismus fronty. V modulu parseru dochází k rozložení příchozích síťových dat na jednotlivé hlavičky protokolů. Takto naplněná datová pole jsou předány do vstupní match+action posloupnosti tabulek, kde dochází ke změně paketu a výběru výstupu. Poté vstupní match+action tabulky předají data do mechanizmu fronty, který implementuje distribuci provozu založenou na konfiguraci ze vstupní match+action posloupnosti tabulek. Fronty umožňují měnit pořadí paketů a tím implementovat například mechanismus Quality of Service (QoS). Výstupní posloupnost match+action tabulek je používáno pro konečnou modifikaci hlaviček protokolů před jejich opětovným složením.



Obrázek 2.2: Grafické znázornění dvou fází chodu jazyka $P4_{14}$ (převzato z [20]).

Složení se děje v posledním modulu (deparser), který zajistí poskládání síťového paketu zpět z její reprezentace ve formě protokolových hlaviček.



Obrázek 2.3: Abstraktní model $P4_{14}$ zařízení (převzato z [15]).

Specifikace formátu hlaviček

Popis struktury protokolu je v jazyce $P4_{14}$ intuitivní. Vše je uvedeno klíčovým slovem *header_type*, za kterým následuje identifikátor dané hlavičky. Do sekce *fields* uvádíme formát jednotlivých polí naší hlavičky ve tvaru: název_pole : šířka v bitech; Jako ukázku můžeme uvést deklaraci hlavičky protokolu IPv4.

```
header_type ipv4_t {
    fields {
        version     : 4;  // šířka v bitech
        ihl         : 4;
        diffserv    : 8;
        totalLen    : 16;
        identification : 16;
        flags       : 3;
        fragOffset  : 13;
        ttl         : 8;
        protocol    : 8;
        hdrChecksum : 16;
        srsAddr     : 32;
        dstAddr     : 32;
    }
}
```

Deklarovaná hlavička IPv4 protokolu demonstruje hlavičku fixní délky, kde jsou předem známé jednotlivé velikosti všech políček hlavičky, a proto je celková velikost vypočítána jako prostý součet bitových velikostí políček. Předešlé řešení pokryje širokou škálu síťových protokolů, ale existují i protokoly, které svoji celkovou délku obsahují v jejich vlastní hlavičce. Jako příklad si uveďme rozšiřující hlavičky protokolu IPv6. Takové vlastnosti některých protokolů se jazyk $P4_{14}$ snaží řešit pomocí definice klíčového slova *length*, kde programátor může definovat výraz pro výpočet celkové délky hlavičky. Všechny proměnné ve výrazu musí být definovány v předcházející sekci *fields* popisované hlavičky. Navíc jazyk $P4_{14}$ nabízí programátorovi možnost ohraničení maximální délky hlavičky pomocí klíčového slova *max_length*. Jako příklad deklarace hlavičky s proměnlivou délkou můžeme uvést výše zmíněnou rozšiřující hlavičku IPv6 protokolu. Délka hlavičky je uvedena v políčku *totalLen* v bajtech.

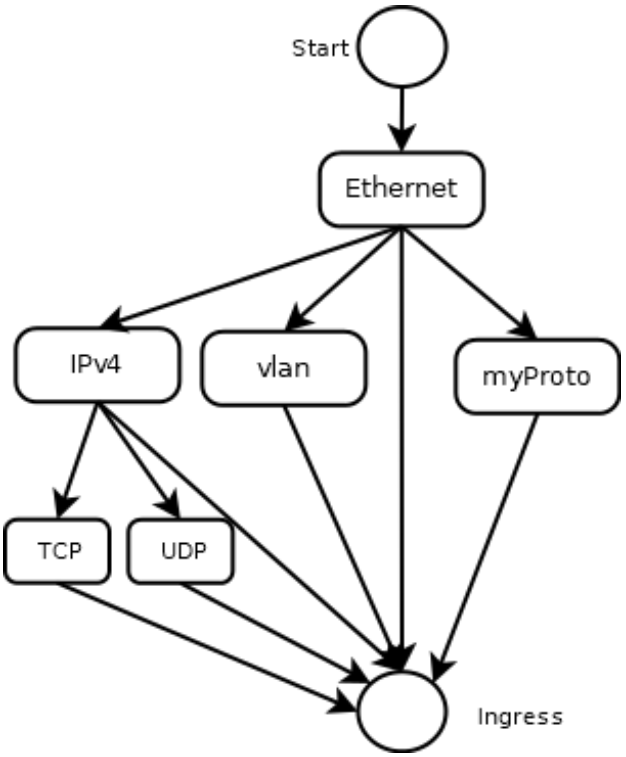
```
header_type ipv6_ext_t {
   fields {
        nextHdr    : 8;
        totalLen   : 8;
        frag       : 12;
        padding    : 3;
        fragLast   : 1;
    }
    length : (totalLen + 1) * 8; // Výpočet délky v bajtech
    max_length : 1024;
}
```

Specifikace parsování

Proces parsování v $P4_{14}$ je implementován jako konečný stavový automat, kde přechod mezi jednotlivými uzly je vždy podmíněn hodnotou již extrahovaného políčka hlavičky a slouží k definici vztahu mezi jednotlivými protokolovými hlavičkami. Přechodem do koncového stavu automatu je spuštěn tzv. kontrolní program. Pro názornost je uvedena definice jednoho uzlu pro parsování ethernetové vrstvy. Schéma konečného automatu znázorňující

strukturu parsování je uvedeno na obrázku 2.4. Pro demonstraci komplexnějšího programu jsou ve schématu vyobrazeny i uzly navíc. Oproti ukázce zdrojového kódu se uzel IPv4 dále větví a umožňuje zpracovat TCP nebo UDP hlavičku. Ostatní uzly nenabízí možnost větvení a odkazují přímo na vstupní bod programu v našem případě *Ingress*.

```
header ethernet eth;
parser parse_ethernet {
   extract(eth);
   switch(eth.ethertype) {
      case 0x8100: vlan;
      case 0x9100: vlan;
      case 0x0800: ipv4;
      case 0xA100 mask 0xF100 : myProto;
      default : ingress;
   }
}
```



Obrázek 2.4: Stavový automat popisující strukturu parsování

Deklarace proměnných jednotlivých typů hlaviček je uvedeno klíčovým slovem *header*. Stavy konečného automatu jsou v jazyce $P4_{14}$ tvořeny klíčovým slovem *parser*, za kterým ihned následuje identifikátor tohoto stavu. V něm nejdříve dochází k extrakci(klíčové slovo *extract*) dat do zadané proměnné. Na základě extrahovaných dat, jejichž část je použita jako argument příkazu *select*, je vybrán následující stav. Výběr může být prováděn i na základě více políček zpracované hlavičky. V tomto případě jsou v argumentu příkazu *select* uvedená políčka oddělena čárkou. Poté je z políček vytvořen binární vektor, který je použit

pro porovnávání se zadanými hodnotami. $P4_{14}$ umí řešit i případy, kdy nechceme použít k porovnání přesnou hodnotu, ale pouze její část. Toho lze docílit pomocí klíčového slova *mask*. Poté je na aktuální hodnotu vektoru aplikována maska a následně je výsledek použit v porovnání. V následující ukázce je na vektor aplikována maska s hodnotou 0xF0F0 a výsledek operace je porovnán s hodnotou 0xc0c0.

```
0xc0c0 mask 0xF0F0 : p_ipv7;
```

Specifikace tabulek

Tabulky v jazyce $P4_{14}$ slouží pro mapování zpracovávaného datového toku na konkrétní akce. Tabulky jsou rozděleny do sekcí *reads* a *actions*. V sekci *reads* jsou uvedeny protokoly a jejich políčka, na které bude tabulka reagovat. Také je zde uveden algoritmus, pomocí kterého se bude dané políčko porovnávat se záznamy v tabulce. Příklad zápisu tabulky je uveden níže.

```
table filter {
    reads {
        ipv4.dstAddr   : exact;
    }

    actions {
        _pass;
        _drop;
    }
}
```

Uvedená tabulka reaguje na protokol IPv4 a k porovnávání používá políčko dstAddr (cílová IP adresa). Vyhledávání v tabulce záznamů je provedeno pomocí algoritmu *exact*. Mapování na jednotlivé uživatelské akce je realizováno podle pravidel, kterými je zařízení nakonfigurováno přes MI32 rozhraní ze softwaru. V případě výše uvedené tabulky je nutné v nahrávaných pravidlech uvést požadované cílové IP adresy a čísla akcí, které budou při jejich detekci spuštěny. Jazyk $P4_{14}$ nabízí pět algoritmů pro porovnávání dat se záznamy v tabulce:

1. **exact**: Hodnota v zadaném políčku zpracovávaného paketu musí přesně odpovídat jednomu ze záznamů v tabulce.
2. **LPM**(Longest Prefix Match): Porovnání probíhá na základě nejdelšího binárního prefixu zpracovávaných dat.
3. **range**: Porovnání zda zpracovávaná hodnota patří do zadaného intervalu.
4. **ternary**: Každý bit má 3 možné stavy: *0, 1* a *X* (don't care) kdy bit označený jako *X* bit je při porovnání hodnoty z paketu a z tabulky ignorován.
5. **valid**: Porovnání je úspěšné, když hlavička se vyskytuje v právě zpracovávaném paketu.

Tabulka záznamů je po spuštění prázdná a musí být naplněna přes konfigurační API námi zadanými pravidly.

Specifikace akcí

Jazyk $P4_{14}$ nabízí sadu vestavěných tzv. primitivní akcí. Navíc také umožňuje uživateli vytvořit si vlastní funkce, které mohou být tvořeny primitivními funkcemi a již dříve definovanými uživatelskými funkcemi.

```
action filter {
    modify_field(ethernet.ethertype, 0x800);
    remove_header(ipv7);
}
```

Výše uvedená uživatelská akce provede odstranění IPv7 protokolu z právě zpracovávaného paketu. Nejdříve nastavíme políčko ethertype ethernetového protokolu na hodnotu 0x800, která je dle standardu IEEE vyhrazena IPv4 protokolu. Poté odstraníme samotný IPv7 protokol. Akce *modify_field* a *remove_header* patří mezi primitivní akce a políčka, která ovlivní, jsou předaná skrze její parametry. I uživatelské akce mohou obsahovat parametry. Ty jsou předány již z match+action tabulek a jejich hodnota může být nastavena během běhu aplikace skrze API společně s nastavením záznamů v tabulkách. Kompletní soupis primitivních akcí je uveden ve standardu jazyka $P4_{14}$ [15].

Specifikace kontrolního programu

Kontrolní program sjednocuje vše předchozí a udává pořadí, ve kterém budou jednotlivé tabulky volány. Volání probíhá pomocí klíčového slova *apply*. Následující tabulky mohou být vybrány pomocí příkazu hit/miss v aktuální tabulce a nebo na základě selekce pomocí příkazu if-else. Příklad takového kontrolního programu můžeme vidět v níže uvedeném kódu:

```
control ingress {
    apply(remove_ipv7);
    if(valid(ipv4))
        {
                apply(filter);
        }else
        {
                apply(drop_packet);
        }
}
```

Výše uvedený program definuje chování celé aplikace. Nejdříve je spuštěna tabulka pro odstranění protokolu IPv7. Poté se spustí tabulka pro filtrování, pokud je ve zpracovávaném paketu detekován protokol IPv4, v opačném případě spustíme tabulku *drop_packet* a paket zahodíme.

2.2.3 Statistiky v $P4_{14}$

Sběr statistik je nedílnou součástí provozování počítačových sítí. Statistiky nám pomáhají při vylepšování síťové infrastruktury, kdy na jejich základě můžeme upravovat topologii jednotlivých částí sítě či posílit její nejvyužívanější oblasti. Pro zajištění větší bezpečnosti dnes nestačí analyzovat datový tok po jednotlivých datových rámcích - paketech. Zmíněný přístup odhalí pouze vybraná nebezpečí, ale útoky, které jsou rozloženy mezi více paketů

nechá bez povšimnutí. Pro komplexnější diagnostiku provozu v síťové infrastruktuře je potřeba vést záznamy, které umějí přesněji odhalit riziko. I standard $P4_{14}$ je vybaven funkcemi pro agregaci informací nad zpracovávaným datovým tokem. V následující sekci bude detailněji popsána realizace statistik v $P4_{14}$. Informace k primitivním akcím stavových pamětí jsou obsaženy v příloze B.

Stavová paměť v $P4_{14}$

$P4_{14}$ umožňuje agregovat informace o datovém toku za využití tzv. stavové paměti. Stavová paměť se skládá z *čítačů*, *měřičů* a *registrů*. Ty jsou uspořádány do polí paměťových buněk, kde jednotlivé buňky mohou být zpřístupněny skrze jméno a index tohoto pole. Dotaz na jednotlivé buňky z důvodu získání či aktualizace její hodnoty lze provádět prostřednictvím akcí, které byly vybrány v tabulkách. $P4_{14}$ rozlišuje tři základní režimy, ve kterých se mohou *čítače*, *měřiče* a *registry* vyskytovat:

global stavová paměť je globální a může být zpřístupněna z kterékoliv tabulky celého $P4_{14}$ programu. Při deklaraci musí být uveden atribut *instance_count*.

static statický režim umožňuje přístup k dané stavové paměti pouze z dedikované tabulky. Při deklaraci musí být uveden atribut *instance_count*.

direct tzv. přímý režim při deklaraci implicitně vytváří jednu buňku stavové paměti pro každý záznam určené tabulky. V režimu direct není třeba pro *čítače* a *měřiče* uvádět primitivní akce, protože aktualizace deklarované stavové paměti se provádí implicitně. V případě *registrů* musí být primitivní akce uvedeny.

V následujících částech budou konkrétněji popsány jednotlivé prvky stavové paměti dostupné v $P4_{14}$.

Čítače

Čítače jsou prvním zástupcem stavové paměti v $P4_{14}$. Níže je uvedena ukázka deklarace *čítače* v režimu static pro tabulku mTag_table jehož paměťové pole obsahuje 512 buněk (vlevo). Napravo je uveden příklad deklarace *čítače* v režimu direct.

```
counter bytes_by_dest {
    type : bytes;
    static : mTag_table;
    min_width: 5;
    instance_count : 512;
    saturating;
}
```

```
counter packets_by_dest {
      type : packets;
      min_width: 6;
      direct : mTag_table;
}
```

Jako atribut *type* je možno zadat jednu z možností uvedených níže a tento atribut určuje chování jednotlivých čítačů v případě, kdy došlo k jejich zacílení explicitně (v případě *static* čítačů) a nebo implicitně (v případě *direct* čítačů). Vlastnosti a názvy jednotlivých typů čítače jsou:

bytes čítač je inkrementován o hodnotu, která odpovídá velikosti paketu v počtu bajtů

packets hodnota čítače je inkrementována s každým příchozím paketem.

packets_and_bytes typ čítače je kompromis mezi předešlými dvěma typy. Čítač je složen z dvou samostatných čítačů, které jsou inkrementovány o hodnotu délky paketu respektive o hodnotu jedna. Typ zajišťuje jednotný přístup k oběma čítačům.

Specifikace jazyka $P4_{14}$[15] umožňuje povolit saturaci požadovaného čítače a nastavit minimální bitovou šířku čítače. Pro čítače je dedikována pouze jediná primitivní akce *count* (viz příloha B).

Měřiče

Měřiče jsou stavové objekty, které měří rychlost datového provozu buď to v paketech nebo bajtech za sekundu a výsledek je prezentován jako jedna ze tří barev: červená, žlutá, zelená. Barvou je následně označen právě zpracovávaný paket. Specifikace jazyka $P4_{14}$[15] neuvádí konkrétní přístup k implementaci měřičů a jako jednu z možností jejich realizace odkazuje implementaci v RFC2697 [9]. Deklarace měřičů je podobná jako u čítačů:

```
meter meter_name {
    type : bytes | packet;
    direct | static : mTag_table;
    [instance_count] : 512;
    [result]: field;
}
```

Měřiče mohou být odkazovány v primitivní akci pouze pokud se jedná o statický nebo globální měřič. Stejně tak atribut *instance_count* je povolen jen u těchto měřičů. Nový prvek, který nám měřiče přináší, je atribut *result*. Lze jej použít pouze u direct měřičů a určuje políčko, na které má být zaznamenána značka vytvořená měřičem. Měřiče mají podobně jako čítače k dispozici pouze jednu primitivní akci *execute_meter*(viz příloha B).

Registry

Registry jsou posledním zástupcem stavové paměti v jazyce $P4_{14}$. Jedná se o stavovou paměť jejíž hodnota může být zapisována a čtena v uživatelských akcích. Jsou podobné čítačům, ale jejich použití je obecnější. Typickým příkladem užití můžeme uvést příklad signalizace, že první paket daného toku již byl zpracován zařízením. Při detekci daného toku se v uživatelské akci nastaví konkrétní registr jako označený. Informace může být poté uložena jako metadata daného paketu a tak předána do další tabulky pro kontrolu, zda je daný paket označený. Deklarace registrů je podobná jako čítačů:

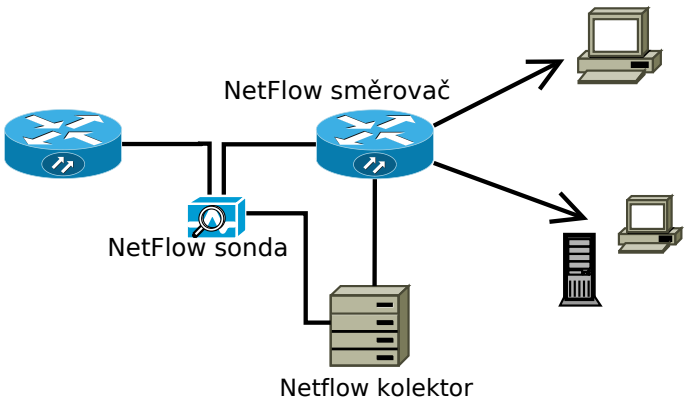
```
register register_name {
    width : 48;
    direct | static : mTag_table;
    [instance_count] : 512;
}
```

U deklarace je nutné uvést bitovou šířku nutnou alokovat pro jednu paměťovou buňku. V případě statických registrů je uveden jejich počet a jméno tabulky, se kterou jsou registry spjaty. U přímých registrů je počet registrů odvozen z počtu záznamů dané tabulky. Požadavek na statické i přímé registry je proveden pomocí primitivních akcí. To je výjimka oproti předchozím stavovým pamětem, kde dotazování na přímě prvky docházelo implicitně. K dotazování je využíváno primitivních akcí *register_write* a *register_read* (viz příloha B).

Popis monitorování síťových toků v jazyce P4$_{14}$

NetFlow je otevřený protokol vyvinutý společností Cisco Systems. Jeho hlavním účelem je monitorování síťového provozu na základě IP toků. IP tok bývá definován jako sekvence paketů se shodnou pěticí údajů: cílová/zdrojová IP adresa, cílový/zdrojový port a číslo protokolu. NetFlow statistiky lze použít pro odhalovaní bezpečnostních incidentů a úzkých míst systému, vylepšovaní stávající architektury či určování poplatků za přenesená data po síti.

NetFlow architektura je typicky složena z jednoho kolektoru a několika exportérů NetFlow záznamů. Exportéry jsou připojeny k monitorované lince a analyzují průchozí pakety. Na základě IP toků generuje NetFlow statistiky, které poté odesílá do kolektoru. Tradiční NetFlow architektura umísťuje exportéry statistik přímo na směrovače. Zmíněný přístup má nevýhodu ve vysoké ceně směrovačů. Lze pořizovat i levnější směrovače schopné pořizovat NetFlow statistiky, ale za cenu zmenšení přesnosti měření, protože levnější NetFlow směrovače pořizují statistiky z každého n-tého paketu. Moderní architektura NetFlow je založena na zapojování pasivních sond na monitorovanou linku. Sondy provoz na lince pouze monitorují a nijak neupravují. Pro doručení dat do kolektoru podle protokolu NetFlow používají dedikovanou linku. Na obrázku 2.5 je uvedeno zapojení pasivní NetFlow sondy i směrovače.



Obrázek 2.5: Ukázka zapojení systému NetFlow

Následující ukázka demonstruje možnost implementace základního počítání NetFlow statistik v P4$_{14}$. Akce *count_netflow_stats* je spuštěna pro každý vstup do tabulky, který nalezne shodu se záznamem v tabulce alespoň v jednom případě. *Čítač* se aktualizuje implicitně pro každý vstup do tabulky. V akci *count_netflow_stats* se poté aktualizuje *registr pkt_bytes* o počet přenášených bajtů a *flags_reg* aktualizuje TCP Flag.

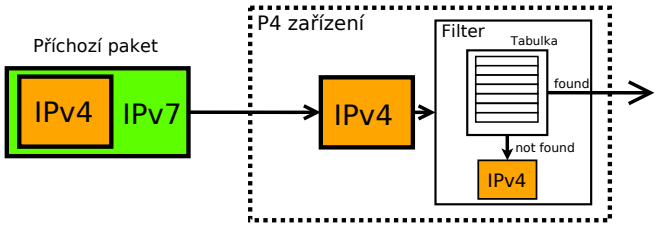
```
// Každý řádek má vlastní registry a čítače
counter pkt_cnt {
    type : packets;
    min_width : 32;
    direct : table_netflow_count;
}

register pkt_bytes {
    width : 256;
    direct : table_netflow_count;
```

```
}

register flags_reg {
    width : 256;
    direct : table_netflow_count;
}

register last_ts {
    width : 256;
    direct : table_netflow_count;
}

table table_netflow_count {
    reads {
        ipv4.srcAddr : lpm;
        ipv4.dstAddr : lpm;
        ipv4.proto : ternary;
        tcp.srcPort : exact;
        tcp.dstPort : exact;
        tcp.flags : ternary;
    }
    actions {
        count_netflow_stats;
    }
    min_size: 512;
}
action count_netflow_stats() {
    // Aktualizace prenesenych bajtu
    register_read(pkt_bytes,met.bytes);
    add(met.bytes,pkt.length);
    register_write(met.bytes,pkt_bytes);
    // Aktualizace timestampy
    register_write(last_ts,pkt.ts);
    // Aktualizace TCP Flagu
    register_read(flags_reg,met.flags);
    bit_or(met.flags, tcp.flags);
    register_write(flags_reg,met.flags);
}
```

2.2.4 Definice nového protokolu v jazyce $P4_{14}$

V následující části je popsána aplikace IPv7 v jazyce $P4_{14}$, na jejíž vývoji jsem se podílel v rámci výzkumné skupiny Liberouter. Zdrojové kódy této aplikace jsou přiloženy v příloze A. Účelem aplikace je demonstrace flexibility jazyka P4 s použitím FPGA obvodů za využití vymyšleného tunelovacího protokolu IPv7. Struktura protokolu, který zabaluje klasický IPv4 protokol, je zachycena v tabulce 2.2. Síťové zařízení pak umí IPv7 protokol odstranit a filtrovat pakety na základě IPv4 adresy. Pro aplikaci byla také vytvořena kontrolní aplikace v jazyce C, která umí dané zařízení nakonfigurovat pravidly pro filtraci a odstranění

protokolu IPv7. Uvedená aplikace byla prezentována na mezinárodní konferenci IEEE FPL 2017 v Ghentu[5].



Obrázek 2.6: Funkcionalita IPv7 aplikace

Bity	1	2	3
0-2	Identifikace		TTL
3-5	Číslo protokolu		Další hlavička
6-8	Kontrolní součet		
9-11	Zdrojová		
12-14	adresa		
15-17	Cílová		
18-20	adresa		

Tabulka 2.2: Předpis protokolu IPv7

2.3 Struktura prostředí překladače $P4_{14}$

Struktura ekosystému okolo vývoje $P4_{14}$ je tvořena samotným překladačem P4 zdrojového kódu do VHDL a knihovnou *libp4dev* poskytující funkce pro komunikaci mezi softwarovými aplikacemi a P4 zařízením.

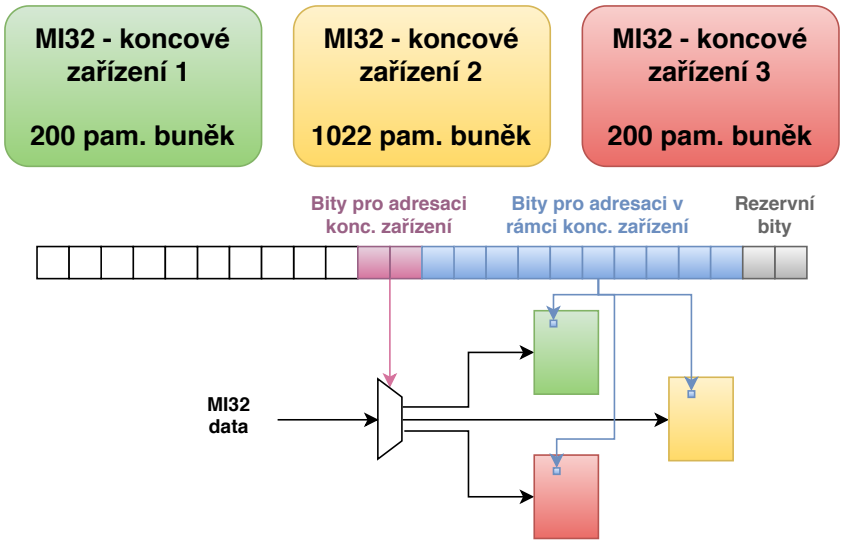
Proces překladu se skládá z více dílčích kroků. Na vstupu je dán zdrojový soubor s popisem v jazyce P4. Z něj je nejdříve pomocí překladače poskytovaného komunitou jazyka P4 vytvořen objektový popis P4 programu v jazyce Python. Vygenerovaná objektová reprezentace se stává vstupem do druhé části překladače zodpovědné za generování VHDL popisu. Objektový popis je použit pro vytvoření grafu, který umožní identifikovat základní stavební bloky P4 kontrolního programu nazvané *Match+Action Groups*. Ty se skládají z bloků *Match+Action Table a Match+Action Router*. První z bloků obsahuje plnou funkcionalitu skládající se z modulu hledajícího nejvhodnější funkci ke spuštění na základě vstupních dat a hodnotách obsazených v záznamech tabulky. Nadále obsahuje modul pro výpočet adresy následné tabulky. Druhý blok, *Match+Action Router*, obsahuje omezenou funkcionalitu vypočítávající adresu následné tabulky či routeru na základě příchozích dat v hlavičkách a metadatech. Dovoluje nám tak realizovat podmíněný příkaz bez nutnosti instanciace vyhledávajícího bloku či bloků realizujících uživatelské akce. Pro každý blok (vyhledávající blok, uživatelskou akci, tabulku) je vygenerován vlastní VHDL zdrojový soubor.

Kromě zdrojových souborů je generován také tzv. devicetree popis. Devicetree je datová struktura sloužící k popisu hardwarových zařízení a jejich adresnímu uspořádání. Každý uzel

tabulky P4 zařízení je popsán offsetem, na kterém se nachází v adresním prostoru, velikostí, jménem a množinou poduzlů. Uzly v sobě obsahují informace potřebné pro korektní chod softwarové aplikace. Data jsou použita v *libp4dev* pro naplnění interních datových struktur, které zajišťují abstrakci zařízení pro uživatele a jeho korektní komunikaci se softwarem.

2.3.1 MI32 sběrnice

Následující text popisují MI32 sběrnici vychází z [3]. MI32 sběrnice je vhodná zejména pro konfiguraci a komponenty, které nevyžadují velkou propustnost sběrnice. Sběrnici lze uspořádat do stromové struktury pomocí komponenty *mi32_splitter*, která dokáže propojit více koncových bodů s jedním zdrojem. Pro korektní adresování do více komponent je nutné znát celkový počet koncových bodů s hodnotou největšího vyžadovaného adresního prostoru. Podle největšího adresního prostoru je vyčleněn požadovaný počet bitů k adresování v rámci paměťových buněk jednotlivých koncových bodů a podle jejich počtu jsou vyčleněny potřebné bity k adresaci. Na obrázku 2.7 je uvedeno schéma adresace MI32 rozhraní. Schéma zachycuje situaci se třemi komponentami, které je potřeba adresovat. Proto jsou pro adresaci koncových zařízení vyhrazeny právě dva bity. Největší pole pro adresaci obsahuje žlutá komponenta s 1022 paměťovými buňkami. Proto je nutné vyčlenit právě 10 adresních bitů pro umožnění adresace všech buněk každého koncového zařízení.

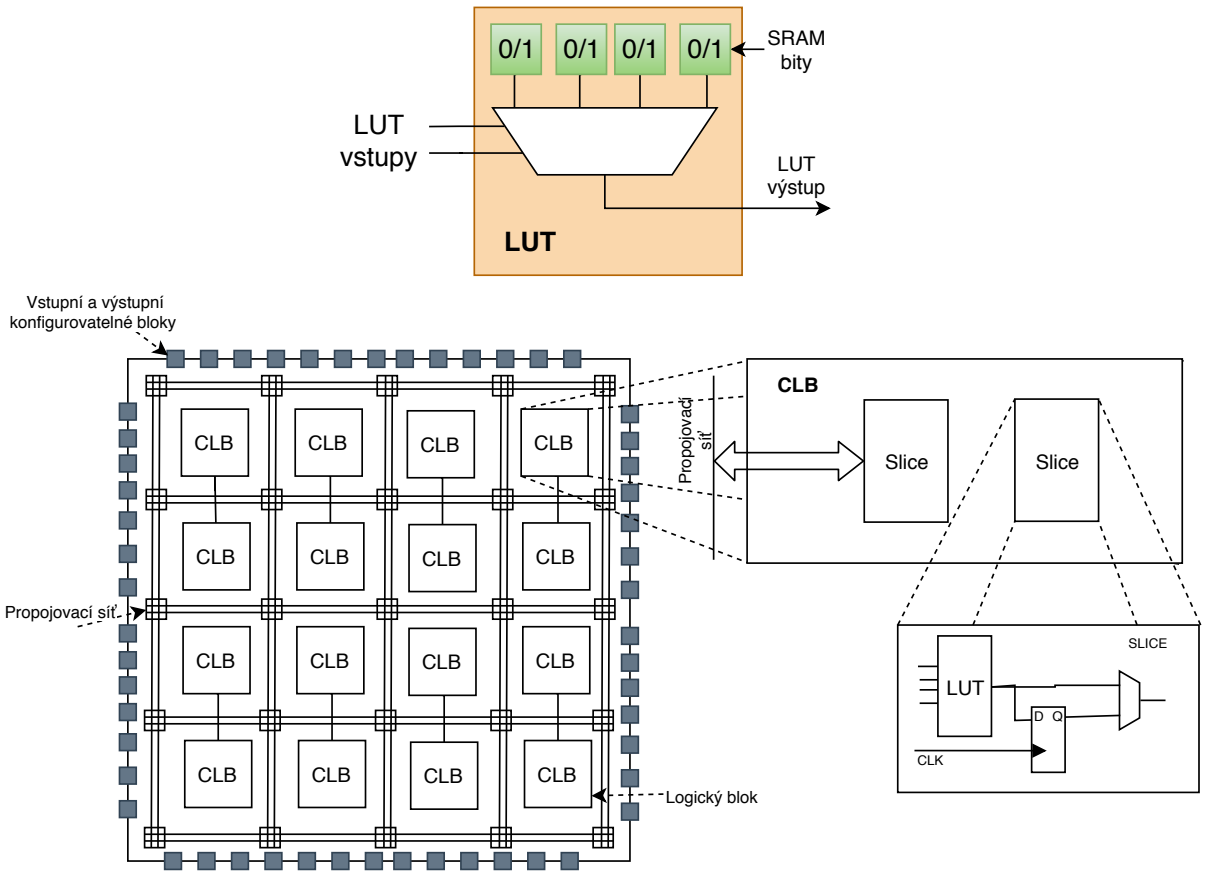


Obrázek 2.7: Adresace MI32 rozhraní

2.4 FPGA

Stavové zpracování je výpočetně náročná činnost. Časté dotazování se na stejné adresy v paměti, výpočet nových hodnot, jejich následná aktualizace a potřeba pracovat neustále s aktuálními daty klade na zařízení realizující stavové zpracování síťového provozu vysoké výkonnostní požadavky. Použití univerzálních procesorů (CPU) je díky jednoduchosti softwarové implementace často volenou variantou. Běžné CPU jsou dle [21] schopné realizovat řešení analýzy dat na rychlostech až 10 Gbps. Pro vysokorychlostní sítě s rychlostmi dosahujícími až 100 Gbps je řešení prostřednictvím CPU výkonnostně nedostatečné. Je

nutné takové zařízení, které je schopné díky využití paralelismu úloh a zřetězení zpracování jednotlivých požadavků vykonávat analýzu velkého objemu dat velmi rychle. Všechny výše zmíněné vlastnosti splňují aplikačně specifické integrované obvody (Application specific integrated circuits, ASIC). Řešení s využitím technologie ASIC dosahuje podstatně lepších výsledků než jeho realizace pomocí softwarové implementace. Vývoj ASIC je bohužel velmi dlouhá, náročná a drahá činnost. Po výrobě žádaného integrovaného obvodu již není možné provádět změny struktury obvodu, proto je proces vývoje ASIC velmi náchylný na chyby v návrhu. Navíc jejich použití není vhodné při podpoře SDN, protože změna požadavku na funkcionalitu sítě často vede k nutnosti takovéto zařízení zaměnit za jiné. Tím se k výše zmíněným požadavkům přidává i požadavek na možnou změnu funkcionality zařízení. Zmíněný nedostatek odstraňují programovatelná hradlová pole (Field Programmable Gate Arrays, FPGA), která nabízí vhodnou alternativu ASIC řešení. FPGA je kompromis mezi rychlostí řešení pomocí ASIC, vůči flexibilnosti softwarového řešení.



Obrázek 2.8: Hierarchická struktura FPGA

Programovatelná hradlová pole jsou tvořena maticí konfigurovatelných logických bloků (Configurable Logic Blocks, CLB), jak je uvedeno na obrázku 2.8. Propojení mezi jednotlivými CLB je realizováno pomocí propojovací matice. Propojovací síť může být lokální či globální. Lokální síť propojuje sousední bloky rychlými lokálními linkami, aby umožnila realizaci složitějších logických funkcí pomocí skládání LUT (bude vysvětleno níže v sekci)

nebo nabízí speciální propoje (tzv. carry-chain) pro tvorbu sčítaček. CLB jsou tvořeny menšími bloky tzv. *slice* buňkami, které jsou využívány jako generátory pro logické funkce a sekvenční logiku. Generátor logických funkcí je realizován jako n-vstupá vyhledávající tabulka (Look-Up Table, LUT) fyzicky implementovaná pomocí paměti SRAM. SRAM bity jsou nakonfigurovány tak, aby plnily požadavky námi zadané logické funkce. Buňka *slice* obsahuje nadále registr, který umožňuje buňku využít pro realizaci sekvenční logiky. Dalšími obsazenými prvky jsou logická hradla a multiplexory pro širší využití *slice* buňky. Na obrázku 2.8 je zachycena hierarchická struktura FPGA. V levém dolním rohu je vyobrazena zjednodušená struktura čipu FPGA s CLB bloky a propojovací sítí. Napravo je uvedeno uspořádání CLB bloku a jeho vnitřních podbloků *slice*. Horní část obrázku 2.8 představuje realizaci LUT. Obrázek zachycuje obecný přístup k realizaci programovatelných hradlových polí, avšak jednotlivé čipy od různých výrobců se mohou lišit a to zejména ve struktuře buňky slice.

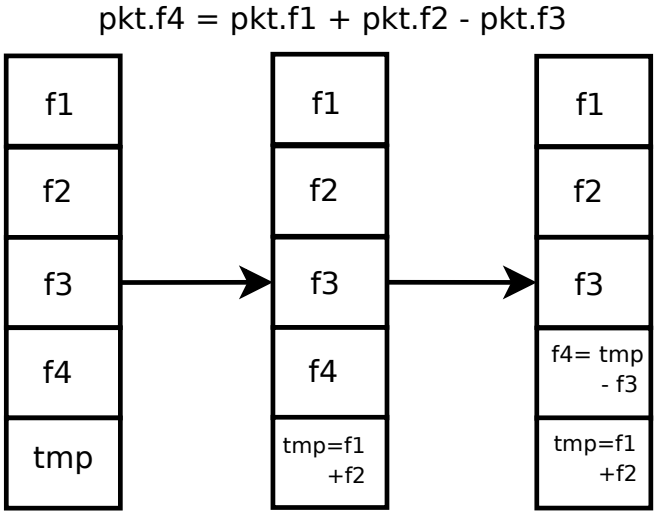
Kromě základních bloků, které jsou určeny pro obecné využití, FPGA čipy poskytují i specializované komponenty a obvody. Pro šíření hodinového signálu jsou k dispozici speciální hodinové obvody. Pro ukládání většího objemu dat čipy poskytují blokové či distribuované paměti typu RAM. Volba implementace paměti závisí na velikosti dat, která jsou nutná uložit. Blokové RAM lze využít na realizaci FIFO paměti. Dále čipy nabízí jednotky pro připojení k externím zařízením pomocí sériových sběrnic (PCI-Express) a rozhraní pro externí paměti či přímo procesory.

Rychlost, kterou jsou FPGA čipy schopné pracovat, se označuje jako pracovní či taktovací frekvence. Ta udává počet taktů hodinového signálu, který čip vykoná, za jednu sekundu. Maximální frekvence čipu je určena časem, který je potřeba na propagaci hodnoty z jednoho registru do následujícího na nejdelší cestě v celém designu (tzv. zpoždění cesty). V případě nastavení vyšší pracovní frekvence se hodnota nestihne uložit do následujícího registru na cestě a celé řešení nebude pracovat dle očekávání. Uvedené fyzikální limity kladou omezení na volbu pracovní frekvence vůči složitosti řešení a neumožňují propojit pomocí globální propojovací sítě dva libovolné bloky CLB na FPGA čipu, kdy vyšší frekvence zmenšuje prostor pro umístění souvisejících logických bloků.

Kapitola 3

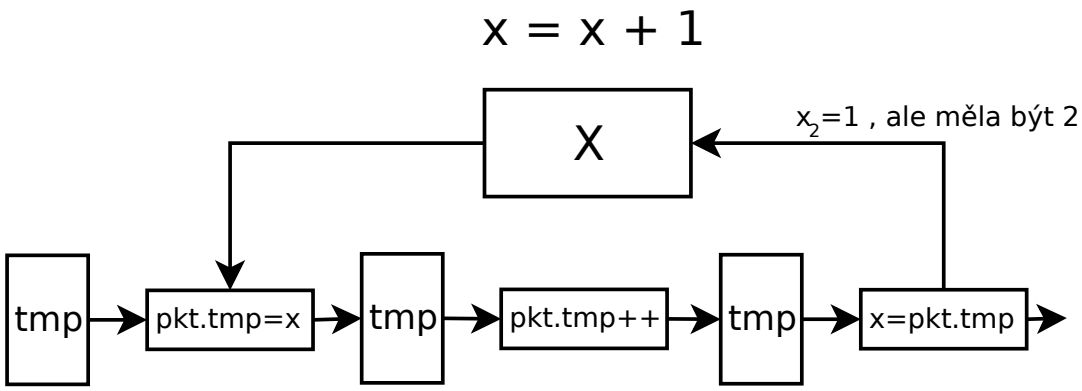
Požadavky na návrh

K analyzování síťového provozu lze využít bezstavové či stavové zpracování. Bezstavové zpracování se omezuje na analýzu jednotlivých datových rámců, zatímco stavové zpracování vede rozsáhlejší informace o datovém toku a analyzuje tok na základě většího časového úseku. Proto stavové zpracování nabízí přesnější analýzu síťového provozu a je jednou z nezbytných funkcionalit, které by měly moderní síťové komponenty nabízet. Pro hardwarově akcelerované síťové komponenty, umožňující provoz vysokorychlostních sítí, je přístup bezstavového zpracování síťového provozu lehce aplikovatelný a umožňuje využití zřetězeného zpracovávání dat. V případě sekvenční logiky jsme takto schopni zpracovávat každý hodinový takt nová data a propustnost našeho zařízení je dána frekvencí designu. Příklad bezstavového zpracování dat je na obrázku 3.1, kdy k výpočtu požadované hodnoty používáme informace pouze z aktuálně zpracovávaného datového toku.



Obrázek 3.1: Ukázka bezstavového zpracování[2]

V případě stavového zpracování je situace odlišná. Potřeba ukládat stavové informace může způsobovat nekonzistentnost dat(viz obrázek 3.2), kdy další data v řetězovém zpracování na kartě nebudou mít k dispozici aktuální hodnotu stavové informace. Proto není možné využívat zřetězené zpracování a musíme zavést atomické funkce pro práci se stavovými informacemi. Atomické sekce zajistí vždy provedení celé operace se stavovou informací a v průběhu vykonávání zamezí přístup k modifikovaným datům.



Obrázek 3.2: Ukázka stavového zpracování[2]

Výše popsaná problematika stavového zpracování společně se znalostí stavových pamětí jazyka P4 identifikovala dílčí cíle, které jsou nutné pro rozšíření stávající bezstavové implementace realizované na platformě FPGA o podporu stavového zpracování. Pro splnění tohoto požadavku je zapotřebí mít k dispozici moduly, které zajistí vhodné uložení dat na omezených zdrojích FPGA čipů a korektní přístup k uloženým údajům, jak z prostředí P4 aplikace tak i ze software pomocí konfiguračního rozhraní MI32. Hlavní sledovanou veličinou je výsledná propustnost zařízení, proto by moduly měly disponovat možností volby vyššího výkonu na úkor spotřebovaných zdrojů. Sadu takto navržených modulů je nadále nutné rozšířit o pomocné komponenty, které zajistí jejich začlenění do celého systému jazyka P4. Jazyk P4 nabízí nespočet kombinací, kterými mohou být stavové paměti vzájemně použity v programech. Proto je nutné vytvořit jednotný mechanismus, který zajistí atomické vykonávání operací nad stavovými paměťmi pro jakýkoliv korektní případ jejich použití. S přidáním stavových prvků do systému vzniká i požadavek na následné rozšíření nástrojů používaných externími aplikacemi pro konfiguraci zařízení.

Kapitola 4

Návrh řešení

V této kapitole bude blíže popsán návrh implementace komponent sloužících k realizaci stavového zpracování v jazyce $P4_{14}$. Jak bylo uvedeno v sekci 2.2.3 k uchovávání statistik v $P4_{14}$ slouží tzv. stavová paměť, která se skládá z *čítačů*, *registrů* a *měřičů*. Cílem práce bylo rozšíření překladače o první dvě konstrukce jazyka $P4_{14}$. Zadání bylo později rozšířeno i o zbývající třetí komponentu *měřičů*.

Pro realizaci stavových prvků je možné využít tři přístupy. Prvním přístupem je využití klopných obvodů v logických buňkách FPGA čipu. Dalším přístupem je použití look-up tabulky. Pomocí obou dvou zmíněných přístupů je možné vytvářet velmi flexibilní paměťové bloky, které jsou vhodné pro realizaci pouze malých pamětí. Paměť je vytvářena z univerzálních logických bloků, které jsou tímto použitím zabrány pro jiné využití. Poslední možnou volbou je využití blokových pamětí RAM, jejichž funkcionalita je přímo určená pro uchování větších objemů dat na FPGA čipu. Jejich použití ušetří univerzálnější buňky pro jiné užití. Nevýhoda používání blokových RAM pamětí spočívá v jejich umístění na FPGA čipu. Zatímco z univerzálních buněk lze vytvořit požadovanou paměť v bezprostřední blízkosti požadované funkcionality, v případě blokových pamětí je fyzické umístění fixováno k určitému místu na čipu a jejich umístění nelze dynamicky měnit. Zmíněný fakt má nepříjemný důsledek. Větší vzdálenost od bloku (využívající BRAM) může zhoršit časování a tím snížit pracovní frekvenci celého zařízení. Problém se zhoršením časováním lze řešit přidáním registrů na kritické cesty, avšak potřeba čekat na vyčtení či zapsání konkrétních dat o hodinový takt déle je nežádoucí z důvodu zmenšení propustnosti celého zařízení.

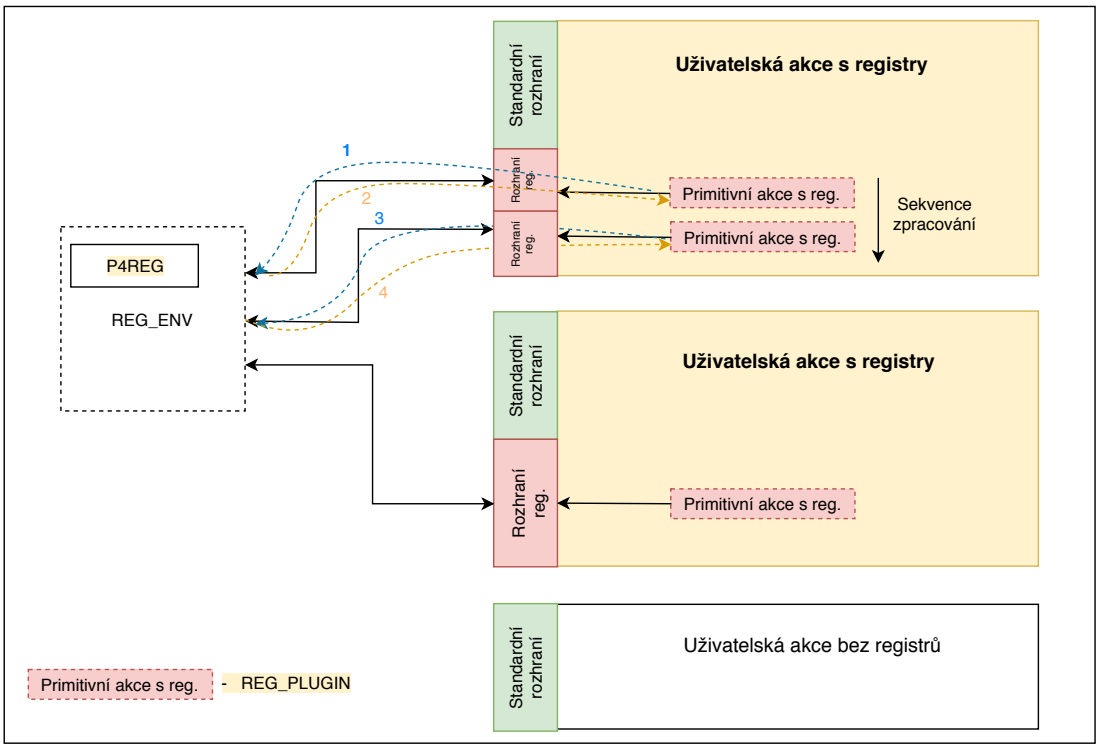
Proto byl při návrhu komponent zvolen kompromis v možnosti volby druhu použité paměti na vytvoření paměťových polí a zodpovědnost finálních vlastností celého zařízení je přesunuta na uživatele, který si volí mezi spotřebovanými zdroji a celkovou propustností.

4.1 Registry

Standard $P4_{14}$ umožňuje velmi různorodé využití registrů, proto bylo nutné návrh rozdělit na vhodné dílčí komponenty, které by tuto flexibilitu použití vhodně reflektovaly. Pro korektní chod stavové paměti bylo nutné zajistit atomicitu jednotlivě prováděných operací v celém návrhu a posloupnost vykonávání primitivních operací v akci.

Obecné schéma celé architektury je zobrazeno na obrázku 4.1. V případě registrů jsou primitivní akce vždy uváděny explicitně a to i pro přímé(direct) registry. Proto je nutné propojit jednotlivé primitivní akce registrů se samotnými registrovými poli. Uživatelské akce využívající registrové primitivní akce mají vygenerované rozhraní pro komunikaci s regist-

rovými poli (obrázek 4.1 - Rozhraní reg.). Registrové rozhraní je vygenerováno pro každou použitou registrovou primitivní akci. Jednotlivá rozhraní obsahují synchronizační a datové signály popisující požadavky primitivních funkcí. Všechny požadavky cílí na jediný modul *P4 Registru* obsahující paměťová pole (P4REG). Přístup k *P4registru* z více míst je sjednocen komponentou *Obálky registrů* (REG_ENV).



Obrázek 4.1: Navržená architektura registrů v rámci jedné tabulky

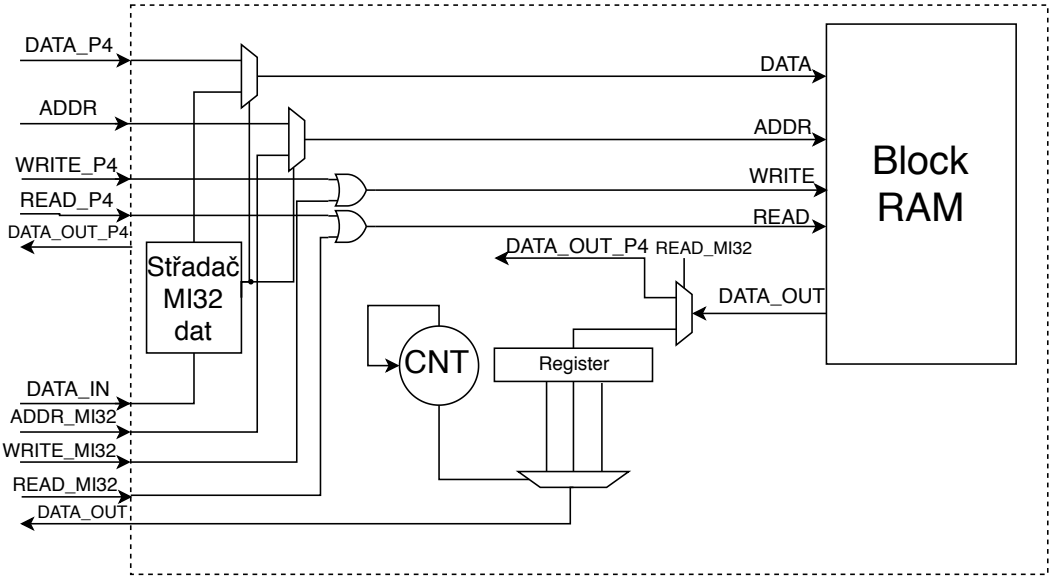
Obrázek 4.1 zachycuje i proces zpracování jednoho průchodu uživatelskou akcí s dvěma primitivními akcemi pracujícími s registry. V prvním kroku (obrázek 4.1, šipka 1) je generován požadavek na stavovou paměť. Zpracování v rámci uživatelské akce je pozastaveno a čeká na odpověď skrz první registrové rozhraní. Současně komponenta *Obálka registrů* detekuje požadavek od rozhraní a označí si jej jako aktivní pro pozdější korektní propagaci výsledků. Poté se *Obálka registrů* dotáže *P4 Registru* na požadovanou paměťovou buňku. Data získaná od *P4 Registru* společně se synchronizačními signály jsou poté *Obálkou registrů* směrovány přes aktivní registrové rozhraní zpět do kontextu uživatelské akce (šipka 2). Příjem potvrzovacích signálů umožní generovat požadavek druhé primitivní akce a zajistí tak sekvenční vykonávání jednotlivých primitivní akcí (šipky 3 a 4). Synchronizační mechanismus zaručuje, že v každý okamžik je aktivní maximálně jedno registrové rozhraní.

4.1.1 P4 Registr

Komponenta *P4 Registr* (obrázek *P4REG*) je základní komponentou celého návrh stavového zpracování v $P4_{14}$ a nabízí dvě možnosti realizace paměti samotné a to pomocí registrových polí a nebo blokových pamětí RAM.

Při vytváření více registrových polí s různou požadovanou cílovou velikostí k jedné tabulce má každá buňka ve zvolené paměti velikost podle největšího požadavku na alokaci.

Volba realizace uložení dat je jádrem komponenty. *P4 Registr* nad jádrem vytváří jednotné rozhraní pro komunikaci se zvolenou pamětí a pro možnost konfigurace ze software pomocí MI32 rozhraní. Na obrázku 4.2 je uvedeno názorné schéma zapojení blokové paměti RAM s možností čtení a zápisu skrze MI32 rozhraní.



Obrázek 4.2: Komponenta registru

Komponenta registru je navržena jako dvou portová a tak, aby na portech A prioritně vyřizovala požadavky příchozí z MI32 a až poté příchozí data z P4 aplikace. Druhá sada portů je již vyčleněna pouze datům z P4 aplikace. V návrhu a implementaci první verze registrů je sada B portů nevyužívána z důvodu aplikace uzamčení zpracování datového toku v celém programu, který zajistí přístup k maximálně jedné paměťové buňce v daný okamžik. Mechanismus zámků bude blíže popsán v části 5.1.

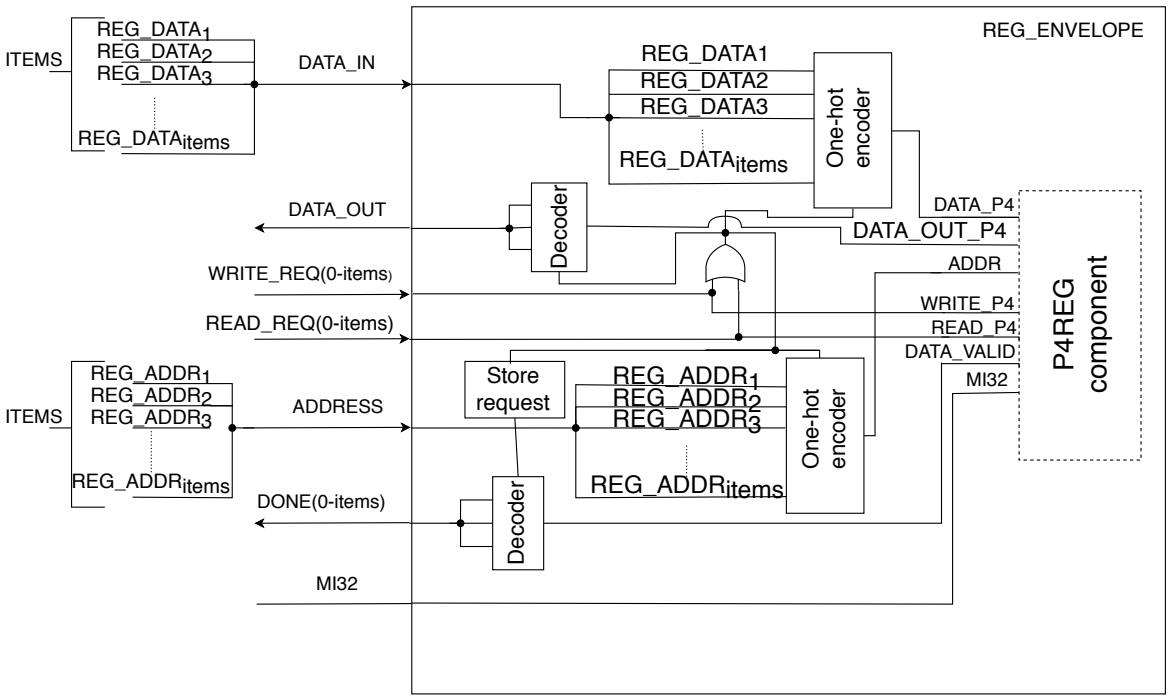
Důležitou rolí komponenty je zajištění správné konfigurace ze softwarové aplikace. Konfigurace probíhá pomocí MI32 rozhraní, které je schopné v jedné transakci přenést nanejvýš 32 bitů. Při deklaraci registrů s velikostí vetší než 32 bitů komponenta postupně skládá transakce až do zaplnění střadače. Skládání dat do střadače je započato při příjmu prvního požadavku na zápis přes MI32 rozhraní . Po obdržení poslední MI32 transakce je vydán požadavek na zápis těchto dat. Při čtení registru s vetší velikostí než 32 bitů jsou data z paměti uložena do pomocného registru, rozdělena na 32 bitové transakce a jednotlivě posílána přes MI32 rozhraní.

Při volbě blokové paměti RAM je možné zvolit pomocí generického parametru vytvoření registrů pro výstupní data. Registr umožňuje zlepšit časování celého řešení, avšak vyvolává potřebu čekat o jeden hodinový takt déle při požadavku na čtení. V překladači se implicitně

výstupní registr nevyužívá, jelikož zařízení po syntéze splňuje časování i při cílené pracovní frekvenci 200 MHz.

4.1.2 Obálka registrů

Díky komponentě *Obálka registrů* (obrázek 4.1 - *REG_ENV*) je možné přistupovat k *P4 Registru* z více bodů. Blokové schéma komponenty *Obálky registrů* je na obrázku 4.3. Komponenta přijímaná data přeposílá *P4 Registru* a zodpovídá za správné delegování potvrzujícího signálu a načtených dat do požadované cílové komponenty, která si dotaz na paměť vyžádala. Operaci *Obálka registrů* provádí pomocí *one hot encoderu* (*Decoder*), který nad vektorem požadavků na čtení či zápis vybere příslušná data z vektoru vstupních dat a adresy. Vstupní a výstupní vektory řídících signálů a dat jsou vytvořeny konkatenací dílčích signálů z vygenerovaných rozhraní. Vektory jsou vytvořeny tak, aby jednotlivé pozice navzájem korelovaly a jeden index patřil vždy k jedné cílové komponentě. V obrázku je uveden pomocný registr *Store request*, který slouží pro zajištění propagace potvrzujícího signálu do koncového bodu odkud požadavek vzešel.



Obrázek 4.3: Komponenta Obálka registrů pro propojení uživatelských akcí se stavovou pamětí

4.1.3 Propojení registrů s primitivními akcemi

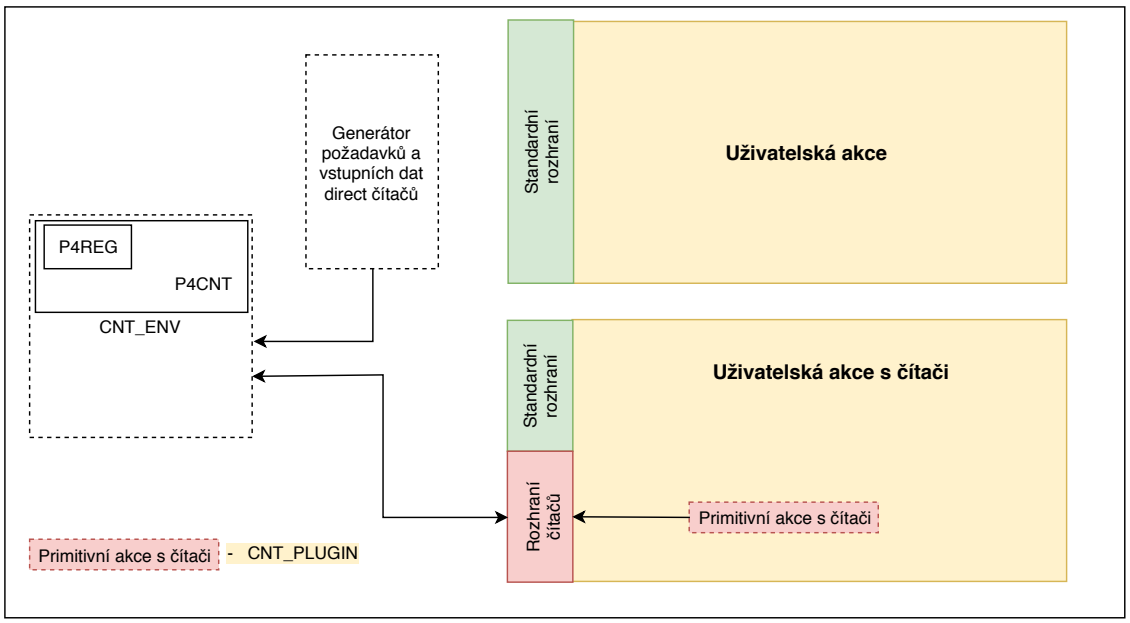
Přístup do registrů a dalších stavových pamětí může být zprostředkován pomocí primitivních akcí. Jejich volání lze provést z jakékoliv uživatelské akce v libovolném počtu. Komponenta *Mapování požadavků* reprezentuje požadovanou registrovou primitivní akci zadanou uživatelem. Komponenta je vytvořena v rámci uživatelských akcí a jejím účelem je namapování korektních vstupů do registrových polí *P4 Registru* pomocí komponenty *Obálky registrů*. Překladačem jsou požadované data namapována na vstupy komponenty *Mapování*

požadavků (obrázek 4.1 - Primitivní akce s reg.). Komponenta pro správný chod tyto data v kontextu celého systému v případě potřeby rozšíří na požadovanou bitovou šířku a namapuje vyžadované signály na konkrétní vstupní/výstupní rozhraní (Rozhraní registrů). Pro každou komponentu *Mapování požadavků* je vytvořeno vlastní *Rozhraní registrů*

4.2 Čítače

Čítače jsou další funkcionalitou nabízenou specifikací jazyka $P4_{14}$ k realizaci stavového zpracování. Čítače zajišťují inkrementaci vybrané paměťové buňky o požadovanou hodnotu. Základním stavebním blokem pro čítač je komponenta registru na obrázku 4.2. Nad *P4 Registrem* byla vytvořena zastřešující komponenta pro implementaci vyžadované funkcionality poskytované dle $P4_{14}$ specifikace.

Obecná architektura celého systému čítačů je zobrazena na obrázku 4.4. Uvažování implicitního přístupu k *direct* čítačům je potřeba respektovat a alokovat pro jejich realizaci nutné zdroje již v prostředí dané tabulky namísto uživatelské akce. Pro správnou funkčnost je nutné řídit a synchronizovat přístup k čítačům z uživatelských akcí a tabulek. Propojení primitivních akcí *count* (viz příloha B) s čítačem je realizováno pomocí speciálně vygenerovaného rozhraní.



Obrázek 4.4: Navržená architektura *static* a *direct* čítačů v rámci jedné tabulky.

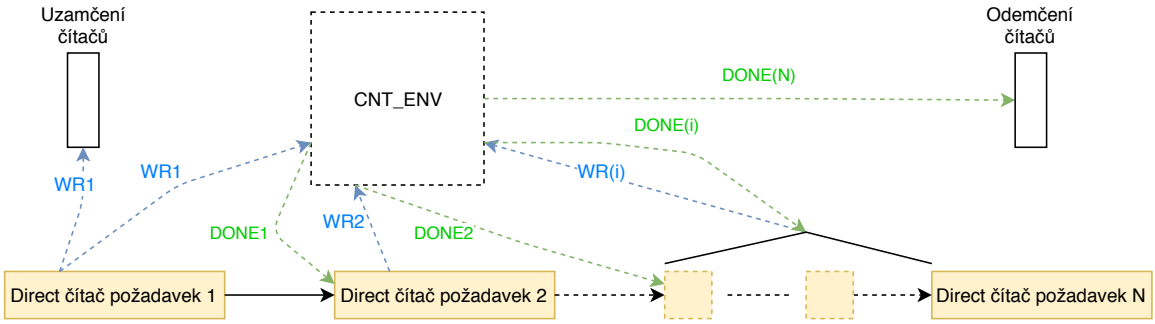
Vyřizování požadavků primitivních akcí probíhá totožně jako v případě registrů. Předchází mu však proces vyřizující požadavky přímých čítačů. Dle jejich deklarace jsou alokovány signály mapující data do komponenty *Obálka čítačů*. Po vyřízení všech požadavků přímých čítačů dochází k vykonávání samotné uživatelské akce.

4.2.1 P4 čítač

Komponenta P4 čítače (obrázek *P4CNT*) je nadstavbou nad základní komponentou *P4 Registr*. Úkolem komponenty je pouze aktualizovat hodnoty, a proto je struktura komponenty prostá. *P4 čítač* vystaví přijatou adresu cílené paměťové buňky na rozhraní komponenty *P4 Registru* a vyčte hodnotu, kterou dle požadavků inkrementuje a opět ji zapíše. Primitivní akce $P4_{14}$ umožňují pouze aktualizovat čítače bez možnosti čtení jejich hodnoty. Proto byla komponenta *P4 čítače* navržena tak, aby každý takt mohla realizovat nový požadavek na inkrementaci požadovaného čítače. Pro zajištění aktualizace korektního čítače komponenta uchovává hodnoty adres v řetězených registrech, kde se hodnoty posouvají společně s vyřizováním požadavků.

4.2.2 Generování požadavků přímých čítačů

Generování požadavků je tvořeno za pomoci sekvenční logiky vygenerované generátorem na základě $P4_{14}$ zdrojového souboru. Jak je zachyceno na obrázku 4.5 pro všechny *direct* čítače jsou vygenerovány komunikační signály. Vyvolání požadavku na aktualizaci čítače je umožněno až po přijetí potvrzení o dokončení předchozí operace. Počet *direct* čítačů není standardem $P4_{14}$ omezen, proto bylo nutné zajistit atomicitu operací pomocí vlastního mechanismu uzamčení. Linka je v případě direct čítačů pozastavena až do okamžiku dokončení všech operací.



Obrázek 4.5: Proces zpracování požadavků na *direct* čítače.

Přístup k *direct* registrům je prováděn z prostředí tabulek a zpracovávání požadavků započne okamžitě po vystavení indexu vybraného záznamu komponentou *search engine*. To umožňuje využít jeden hodinový takt mezi započetím vykonávání uživatelských akcí k vykonání požadavku nad *direct* čítači. Proto se za určité konfigurace čítačů jeví použití jednoho přímého čítače jako operace nemající vliv na propustnost zařízení. Požadavek se podaří vyřídit během jednoho hodinovém taktu a nedojde k pozastavení linky.

4.3 Měřiče

Měřiče jsou posledním druhem stavové paměti nabízeným standardem $P4_{14}$. Jejich funkcionalita není standardem definována. Implementace je založena na základě jednoho ze speci-

fikací nabízených referenčních řešeních popsaného v RFC 2697 [9]. To popisuje měřiče jako síťový prvek, jehož jádro je tvořeno dvěma koši (anglicky *buckets*) s kapacitami značenými jako *Committed Burst Size* (dále CBS) a *Excess Burst Size* (dále EBS). Koš obsahuje *žetony*, které jsou spotřebovávány na základě přijatého síťového provozu. Každý koš je při začátku měření provozu inicializován na svoji maximální kapacitu danou parametry CBS či EBS. Posledním důležitým parametrem měřičů je tzv. Committed Information Rate (dále CIR) udávající počet žetonů přidaných do každého z košů za sekundu. Pro měřiče existují dva režimy, v kterých se mohou nacházet. První z nich se nazývá *Color–Blind* (bezbarvý), který předpokládá, že příchozí provoz není barevně označený a výsledek je určen pouze na základě aktuálního stavu košů. Druhý, *Color–Aware*, předpokládá, že příchozí datový tok je již barevně označený a výsledek je poté vypočítán dle aktuálního stavu košů a barevného označení vstupu. V mé práci jsem se zaměřil na realizaci prvně zmíněného *Color–Blind* režimu, proto bude v další části detailněji popsán pouze tento režim.

Chování měřičů je tedy dáno aplikovaným režimem měřičů a aktuálními hodnotami košů Tc a Te, která jsou na inicializovány na maximální hodnotu Tc(0) = CBS, Te(0) = EBS. Hodnoty Tc a Te jsou poté aktualizovány CIR-krát za sekundu následovně:

- Pokud je Tc menší než CBS, je Tc zvýšeno o hodnotu jedna, jinak
- pokud je Te menší než EBS, je Te zvýšeno o hodnotu jedna, jinak
- nedochází k inkrementaci ani jednoho z košů.

V režimu *Color–Blind* je výsledek měřiče, v čase t příchodu paketu o velikosti B, určen následovně (pro měřiče typu *bytes*: X = B, v případě *packets* typu: X = 1):

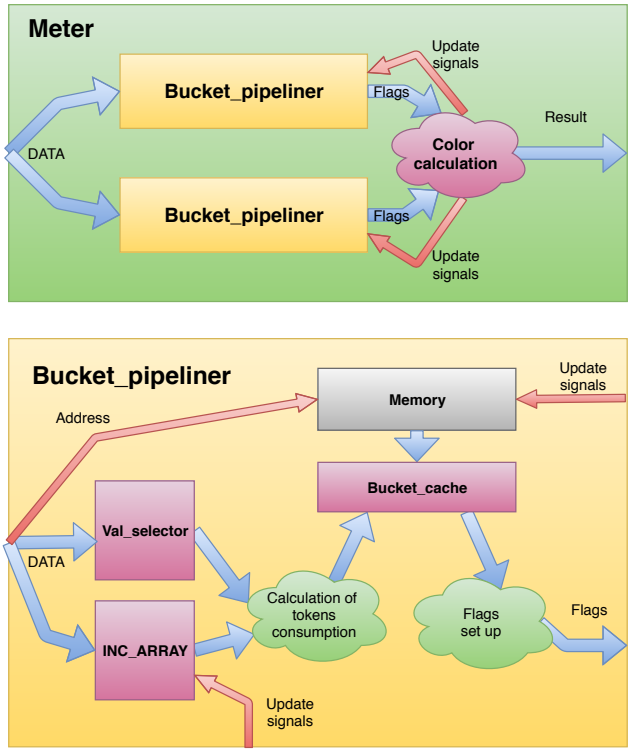
- Pokud Tc(t)-X >= 0, paket je označen zelenou barvou a Tc je zmenšeno o X, jinak
- pokud Te(t)-X >= 0, paket je označen žlutou barvou a Te je zmenšeno o X, jinak
- paket je označen jako červený a hodnoty Tc a Te nejsou modifikovány.

Výše popsané chování vyžaduje komplexnější návrh než v případě předchozích stavových pamětí. Komponenta měřiče byla navržena tak, aby byla schopna zpracovávat nový požadavek každý hodinový takt. Schopnosti zpracovávat data každý hodinový takt bylo dosaženo rozdělením komponenty do menších částí starající se o specifické části algoritmu měřičů či o zajištění datové konzistence.

4.3.1 Struktura měřičů

Struktura komponenty měřiče se skládá z dvou jednotek *bucket pipeliner* a pomocné logiky, jak je uvedeno na obrázku 4.6 v horní části. Z důvodu minimalizace latence jsou komponenty *bucket pipeliner* zapojeny paralelně. Úkolem komponenty je vypočítat výslednou barvu na základě přijatých příznaků a podle přijatých příznaků určit koš, který bude aktualizován.

Bucket pipeliner je stěžejní částí celého návrhu měřičů. Komponenta zastřešuje funkcionalitu prováděnou nad jedním košem. Jejím úkolem je pomocí dílčích komponent zajistit konzistenci dat během celého výpočtu s ohledem na umožnění přijmu nových požadavků každý hodinový takt. Komponenta se navíc stará o korektní aktualizaci hodnot jednotlivých košů, aby hodnota v nich uložena nebyla nikdy větší než je zadaná maximální kapacita. Struktura komponenty *bucket pipeliner* je uvedena na obrázku 4.6 v dolní části včetně schématického zapojení dílčích komponent.



Obrázek 4.6: Struktura komponenty měřiče

Value selector je komponenta, která se stará o detekci mnohonásobného přistoupení na jednu paměťovou buňku. V popsaném případě jsou jednotlivé hodnoty obsažené v dotazech na danou adresu sčítány pro zajištění datové konzistence v následujícím zpracování. Sčítání je prováděna pomocí multiplexoru, který cyklicky rozděluje příchozí data do 4 kanálů, kde tři z nich obsahují registry pro uchování hodnoty. Čtvrtý kanál slouží pro vyřešení přetečení tohoto cyklického bufferu. V případě čtyř bezprostředně za sebou jdoucích dotazů na jednu paměťovou buňku je výsledek okamžitě propagován na výstup do pole sčítaček. V případě detekce přetečení je aktuální součet uložen do pomocného registru, aby jej bylo možné v případě dalšího dotazu na stejnou adresu sečíst s aktuálním vstupem a uložit do některého z kanálů pro zajištění konzistence dat. Každý datový kanál obsahuje multiplexor, který mapuje na výstup korektní data. Pokud je daný kanál aktivní, tak jsou výstupní data tvořená z hodnot uložených v registrech. V případě neaktivního datového kanálu je na výstup přiveden vektor s nulovou numerickou hodnotou. Aktivita datového kanálu je určena řídícím vektorem, který je generován vnitřní jednotkou zvanou *Address pipeliner*.

Increment array komponenta se stará o inkrementaci jednotlivých košů zadaného dle parametru CIR. Pro každou položku měřiče je alokována jedna paměťová buňka obsahující časovou známku poslední aktualizace daného měřiče. Při příchodu požadavku je daná hodnota přečtena a porovnána s aktuální časovou známkou. Dle jejich rozdílu a parametru CIR je výsledně vypočtena hodnota, kterou je nutné přičíst do daného koše. Nová časová známka je aktualizována pouze v případě aktualizace hodnoty měřiče nebo plného koše.

Z důvodu zajištění konzistence dat i v případě rychlého vzájemného přepínání adres na ty, které byly vypočítány, ale jejich hodnoty se nestihly aktualizovat, byla implementována komponenta *bucket cache*. Jednotka si uchovává pro každý požadavek dvě hodnoty. První je hodnota přečtená z paměti, zatímco druhá uchovává hodnotu posledně vypočtené

hodnoty pro případnou aktualizaci. V případě více v řadě jdoucích dotazů na jednu paměťovou buňku je využívána první z hodnot. Při dotazu na adresu, která je obsažena v *bucket cache*, ale nebyla bezprostředně před tím dotazována, je využita druhá z hodnot obsahující potenciálně nejaktuálnější hodnotu dané buňky. Záznamy v *bucket cache* mají omezenou dobu platnosti. Doba je zvolena tak, aby po jejím uplynutí bylo zaručeno, že na vstupu bude k dispozici již aktualizovaná hodnota. Po uplynutí doby platnosti jsou záznamy označeny jako nevalidní a jejich paměť může být použita pro nové záznamy.

Paměť je realizována jako u předchozích stavových pamětí pomocí komponenty *P4 Registru*. Nyní již není uživateli umožněno volit mezi různými druhy realizace paměti, ale je poskytnuta implementace pouze v BRAM. Ta je zvolena z důvodu nižších nároků na zdroje FPGA a existujícímu řešení napojení MI32 sběrnice. Vliv dlouhé přístupové doby je minimalizován pomocí komponent popsaných výše.

Výpočetní logika, na obrázku 4.6 v dolní části značena zeleně, vytváří z výsledků komponent *Increment array* a *Value selector* hodnotu, která má být odečtena od počtu žetonů uložených v koši. V případě takto vypočtené záporné hodnoty vliv CIR parametru převýšil hodnotu, která má být odečtena a jedná se o doplnění žetonů do koše. Následně je hodnota odečtena od počtu žetonů. Pokud je výsledkem kladné číslo, jeho hodnota je následně vystavena na výstup společně s aktivním příznakem aktualizace.

4.4 Shrnutí

V předchozích sekcích byly detailně popsány komponenty realizující stavové paměti v $P4_{14}$. Všechny jsou vystavěny kolem komponenty *P4 Registru* zajišťující volbu implementace paměti a ošetření korektního čtení a zápisu pomocí 32 bitových transakcí z MI32 rozhraní. V případě čítačů a registrů je implementace podobná a je kladen důraz na minimalizaci spotřebovaných zdrojů na čipu. V obou případech byly vytvořeny obálky umožňující přístup k paměťovým polím z více míst. Všechny paměťové pole jsou umístěny do jedné paměti BRAM. Vytvořené řešení má negativní vliv na výslednou propustnost zařízení v případě velkého počtu dotazů na konkrétní druh stavové paměti.

Implementace měřičů je řešena odlišně. Pro každé paměťové pole přičleněné k měřičům je vytvořena, z důvodů velké latence, vlastní instance komponenty měřiče. Pro zajištění možnosti přijímaní nového požadavku každý hodinový takt byla komponenta rozdělena do dvou paralelně zapojených komponent starajících se o přístup k jednotlivým košům. Každá z komponent je rozčleněna do více menší zřetězených jednotek starajících se o určitou část výpočtu korektní výstupní hodnoty koše.

	LUT	**FF**	**Bram)**	**f(MHz)**
Value selector	47	43	0	438.6
Increment array	195	162	0.5	498,7
Bucket cache	375	162	0	351.7
Bucket pipeliner	639	504	1	245.8
Měřič	1378	1004	2	232.9

Tabulka 4.1: Potřebné zdroje pro subkomponenty měřičů

V tabulce 4.1 jsou uvedeny zdroje spotřebované jednotlivými subkomponentami měřiče pro 128 paměťových buněk o velikosti 8 bitů. Poslední dva řádky ukazují komponenty, které v sobě obsahují předchozí jednotky. Samotné jednotky jsou schopné dosahovat vysoké

pracovní frekvence. Při spojení komponent výsledná pracovní frekvence klesá. Největší vliv na snížení pracovní frekvence má komponenta *Bucket cache*, protože vyžaduje paralelní vyhodnocování rovnosti mnoha adres a další konstrukce, které jsou náročné na splnění zadaného časování při integraci do celé komponenty měřiče.

Kapitola 5

Integrace stavových pamětí do systému

Z důvodu přehlednosti a dodržení politiky psaní zdrojového kódu ve vývojovém týmu kolem jazyka P4 byly komponenty, popsány v sekcích 4.1, 4.2 a 4.3, rozděleny do zdrojových souborů pro deklarace entit, kde byl název souboru rozšířen o koncovku *_ent*, a architektury komponenty. Závislé komponenty byly sdružovány do společných adresářů. Pro každou komponentu byl vytvořen podadresář pro skripty sloužící k syntéze a nejkritičtější komponenty nebo ty na nejvyšší úrovni byly rozšířeny i o podadresáře pro vykonání simulace.

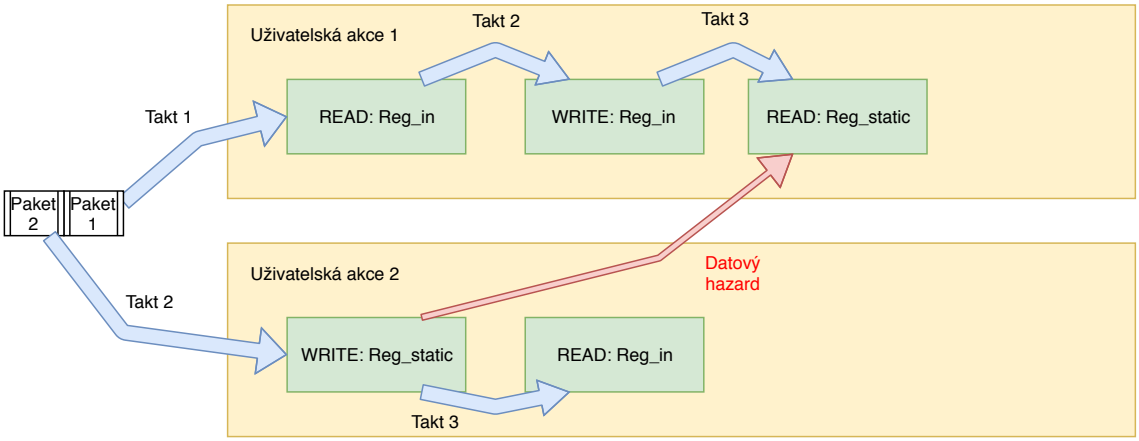
Vytvořené komponenty bylo nutné začlenit do existující architektury, která byla popsána v sekci 2.3. Integrace probíhala zejména ve třídách pracující s tabulkami a uživatelskými akcemi. Byl vytvořen blok speciálních metod, které rozšiřovaly dosavadní řešení o generování pomocných rozhraní pro stavové prvky. Implementované metody byly navrženy obecně z důvodu maximální znovu využitelnosti při instanciaci různých druhů stavové paměti. Překladač byl rozšířen o shromažďování informací stavového zpracování a jejich využití je podrobněji popsáno v sekci 5.2. V sekci 5.1 je detailně popsáno řešení atomických operací nad stavovou pamětí s řešením sekvenčního vykonávání uživatelských akcí. Pro úplnost řešení došlo k rozšíření firmwarové funkcionality, jehož realizace je uvedena v sekci 5.3.

5.1 Řešení atomických operací nad stavovou pamětí

Atomické provedení operací nad stavovými objekty je základem problematiky stavového zpracování. Dle specifikace standardu $P4_{14}$ jazyka musí být zajištěno atomické provedení jednotlivých operací. Při implementaci musela být respektována struktura jazyka, zejména rozdělení do uživatelský akcí a match+action tabulek, v rámci kterých dochází k využívání stavové paměti.

5.1.1 Match+Action tabulky

Běžné P4 zařízení je schopné každý hodinový takt na vstupu zpracovat nově příchozí paket. Při realizaci stavového zpracování je nutné zajistit jednotlivým datovým tokům korektní přístup k stavovým prvků dle pořadí, v kterém přišly do zařízení. Před integrací stavového zpracování do systému P4 byly uživatelské akce navrženy tak, aby dokázaly vykonat svoji činnost během jednoho hodinové taktu. S použitím přístupu ke stavové paměti v rámci uživatelské akce se doba prodlužuje obecně na N hodinových taktů dle zvolených primitivní akcí.



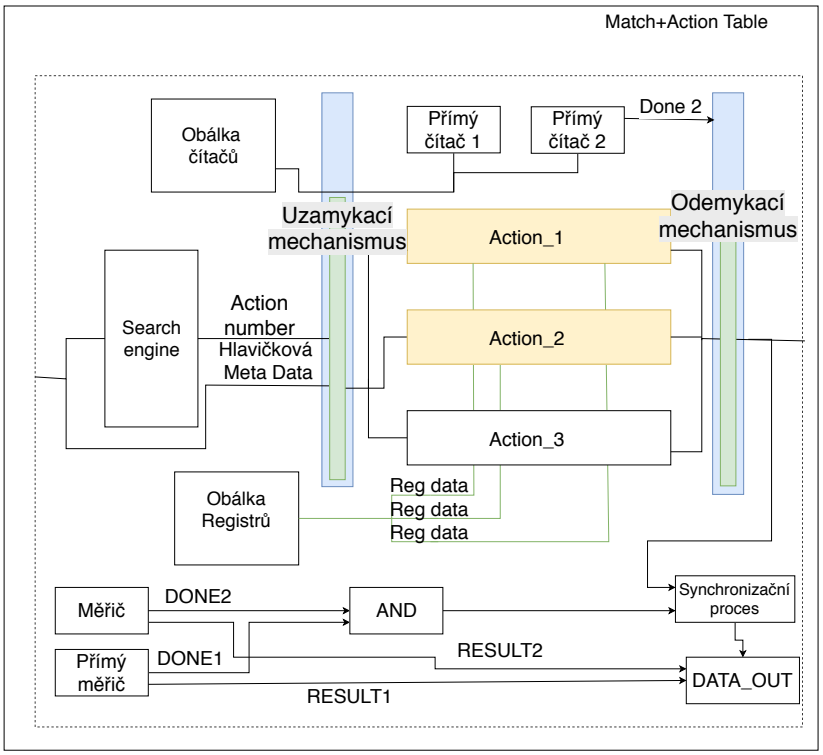
Obrázek 5.1: Vznik datových hazardů

Jak je uvedeno na obrázku 5.1 doba zpracování požadavků na stavovou paměť může vést k nekorektnímu čtení dat. *Paket 1* nepřečte z registru *REG_static* data, která byla v něm umístěna při příchodu do kontextu tabulky. Uložená data jsou pozměněna *paketem 2* ještě dříve, než je *paket 1* dokáže z registru *Reg_static* přečíst. *Paket 2* je totiž namapován na uživatelskou akci 2, během které dříve přistoupí k dané paměti.

Správná práce se stavovými prvky vyžaduje eliminaci datových hazardů. Pro jejich odstranění bylo implementováno uzamykání přístupu k uživatelským akcím během zpracovávání akce, která obsahuje primitivní akce dotazující se na stavovou paměť. Mechanismus zamykání je uveden na obrázku 5.2. Mechanismus zamezuje vzniku datových hazardů. Špatnou vlastností popsaného řešení je zhoršení propustnosti tabulky. Ve výše uvedeném příkladě na obrázku 5.1 by scénář příchodu *paketu 1* vyvolal uzamčení akcí a *paket 2* by byl zpracováván až ve čtvrtém hodinovém taktu.

Mechanismus uzamčení přístupu k uživatelským akcí bylo nadále nutné rozšířit o uzamykání vstupu jiného paketu po zpracování komponentou *search engine* v případě využití *direct čítačů* či *měřičů*. Jak je uvedeno v sekci 4.2.2 dotazování se provádí z prostředí tabulky, proto byl vytvořen i mechanismus pro tuto posloupnost dotazů, aby operace pracovaly se správným paketem. Další paket je zpracováván pouze v případě odemčených obou dvou výše zmíněných mechanismů. Oba zmíněné mechanismy nelze obecně sloučit do jednoho. Jejich rozdělení umožňuje uzamčení tabulky vždy na nejkratší potřebnou dobu.

Na obrázku 5.2 je zachycena struktura match+action tabulky s třemi uživatelskými akcemi, kde akce s názvem *Action_1* a *Action_2* využívají primitivní akce pro práci se stavovou pamětí. *Action_1* využívá právě dvě akce a *Action_2* jednu. V tabulce probíhají i dva dotazy na *direct* čítače. Dva mechanismy zámků umožní v případě vybrání akce s názvem *Action_3* plynulý průchod paketu tabulkou. Nedochází k uzamčení z důvodu využití akce s dotazy na stavovou paměť (obrázek 5.2 zelená přepážka). V tomto případě paket projde tabulkou bez zdržení díky využití jednoho volného taktu před zahájením vykonávání uživatelských akcí, jak je uvedeno v sekci 4.2.2.



Obrázek 5.2: Schéma Match+Action tabulky při použití primitivních akcí pracujících se stavovou pamětí

Kvůli velké latenci komponenty měřičů jsou statické měřiče vyčleněny zcela mimo akce do kontextu tabulky. Jejich zpracování probíhá paralelně s vykonávání uživatelské akce a výsledek je přiřazen do cíleného políčka až na konci zpracování v dané tabulce. Proto není možné pracovat s výsledky měřičů v rámci tabulky, ve které byly vytvořeny. Pro každý měřič je vytvořena vlastní instance pro maximalizaci propustnosti zařízení. Současné řešení je omezeno na použití maximálně dvou dotazů na jedno pole měřičů z důvodu nepotřeby alokace dalšího uzamykacího mechanismu pro měřiče. Dva dotazy lze vyřídit v rámci kontextu jednoho paketu, protože měřice stejně jako přímé čítače využívají jednoho volného taktu při zpracování v tabulce a druhého dostupného během vykonávání uživatelské akce.

5.1.2 Uživatelské akce

Existující řešení uživatelských akcí bylo bezstavové a nebylo připraveno na stavové zpracování. Podporované primitivní akce dovolovaly, aby uživatelské akce byly vykonány během jednoho hodinové taktu. Dle specifikace [15] je nutné zajistit sekvenční posloupnost vykonávání primitivních akcí. V případě akcí pracujícími se stavovými pamětmi bylo nutné pozastavit vykonávání uživatelské akce a uložit si data to registrů. Na vzorkovaná data bylo pak zapotřebí korektně namapovat výsledek akce a předat je k dalšímu zpracování v rámci uživatelské akce. V překladači bylo zapotřebí měnit instanci třídy reprezentující aktuálně používané rozhraní, aby docházelo k mapování korektních signálů s aktuálními daty do primitivních akcí. Generování nového rozhraní bylo umožněno pomoci již implementované metody *prepare_interface* umožňující modifikaci rozhraní přidáním prefixů dle naší volby.

5.2 Výpis informací

Stavové zpracování v některých případech způsobuje snížení propustnosti síťového zařízení. O této skutečnosti je vhodné informovat uživatel a poskytnout mu celkové hlášení o předpokládaném zpomalení linky. V překladači dochází k zpracování informací o použitých stavových prvcích a jejich vzájemné struktuře. Výsledné zpomalení odpovídá použitým primitivním akcí dotazující se na stavovou paměť a realizaci implementace registrového pole. Překladač vyhodnotí, která kombinace primitivních akcí způsobuje největší zpomalení zpracovávání datového toku. Z takto získaných informací je poté vytvořeno přehledné hlášení. Příklad generovaného hlášení je uveden níže.

```
**********************************************************************
* Stateful processing statistics
**********************************************************************
Selected method of implemntation is BRAM

The biggist delay is caused by:
Action     :  add_vlan_tag
In table   :  table_ipv4_filter

Delay      :  +8 clock periods per packet

[Warning] Expected loss of performance: 88 %

Example:
Original device rate without registers: 200 Mpps
Expected device rate with registers: 25 Mpps
Link rate 100 Gbps:
Maximum packet rate(the smallest possible packet 64B):   195 312 500 packets/s
Minimum packet rate(the largest possible packet  1518B):   8 234 519 packets/s
Maximum L2 rate(the smallest possible packet 64B):   12.8 Gbps
Minimum L2 rate(the largest possible packet  1518B): 99 Gbps
```

Výpis informací a výsledky měření.

Výpis stavového zpracování obsahuje jméno uživatelské akce a tabulky způsobující největší zpomalení. Výsledek uvádí kolik je potřeba hodinových taktů pro zpracování paketu procházející touto uživatelskou akcí. Výpis obsahuje i demonstrační příklad očekávaného chování zařízení pracující na frekvenci 200 MHz v extrémním případech provozu na lince s datovou šířkou 100 Gb/s, kde všechny pakety jsou cíleny právě na zpoždění nejnáchylnější akci. Pakety mají vždy stejnou velikost 64 či 1518 bajtů. Výsledné hodnoty byly experimentálně ověřeny v testovacím prostředí Spirent [7].

5.3 Konfigurace

Konfigurační knihovna *libp4dev* je nezbytnou funkcionalitou systému P4. Je napsána v jazyce C a pro každý stěžejní stavební prvek jazyka P4 jsou vytvořeny vlastní moduly. Proto bylo nutné knihovnu a devicetree popis rozšířit o podporu čítačů a registrů.

5.3.1 Realizace stavových pamětí a devicetree

Jak bylo nastíněno v sekci 2.3 každý uzel devicetree popisu sdružuje data popisující jeho vlastnosti jako je velikost či počet záznamů v tabulce. Pro adresaci je uveden offset v rámci P4 zařízení.

Pro každý druh ze stavových pamětí byl zvolen přístup vytvoření vlastního devicetree uzlu. Uzel je v hierarchii P4 zařízení vložen do navázané tabulky. Uzel je umístěn na stejnou úroveň jako uzly popisující záznamy a uživatelské akce tabulky. Dle zadaného druhu paměti je uzel pojmenován jako *registers* či *counters*. Pro každé registrové pole je vytvořen vlastní poduzel, který popisuje offset prvního prvku a hodnoty, které jsou vygenerovány z P4 zdrojového kódu.

V případě alokace více polí registrů či čítačů v rámci jedné tabulky jsou jednotlivá pole sdružována do společné BRAM paměti. V BRAM paměti jsou umístěny sekvenčně za sebou. Při vytváření více registrových polí s různou požadovanou cílovou velikostí k jedné tabulce má každá buňka v BRAM paměti velikost podle největšího požadavku. Pro čítače typu *packets_and_bytes* je vytvořeno pole o velikosti dvakrát větší, než je zadána ve zdrojovém souboru. První část je vyčleněna pro paměťové buňky typu *packets* a druhá je určena pro ukládání dat čítačů typu *bytes*.

5.3.2 Knihovna libp4dev

Knihovna *libp4dev* je implementována v jazyce C a jejím prostřednictvím dochází ke komunikaci mezi softwarem a samotným FPGA čipem. Existující knihovna disponovala funkcemi pro namapování interní struktury na popis zařízení umístěného v souboru *devicetree*. Knihovna byla rozšířena o práci se stavovými prvky. Do struktury *p4device* popisující P4 zařízení bylo přidáno pole *p4counter_t* obsahující informace o čítačích použitých v P4 programu. Byly implementovány moduly *p4counter* a *p4register* obsahující metody pro práci se stavovou pamětí. Jejich prostřednictvím lze stavovou paměť vyčítat, zapisovat, resetovat a získávat její podrobnější popis.

Při vyčítání a zapisování dat do/ze stavové paměti je zapotřebí respektovat fyzickou implementaci paměťových buněk na FPGA čipu, protože její velikost je zvolena dle největšího požadavku na daný typ stavové paměti v tabulce. Proto při zápisu dat je nutné poslat přes MI32 rozhraní užitečná data rozšířena nulovými bity až do požadované velikosti. U čtení je nutné vyčíst celou paměťovou buňku. Knihovna také respektuje strukturu čítače typu *packets_and_bytes* (viz 5.3.1) a korektně s ní pracuje. Výpočet adresy registru na indexu i pro MI32 sběrnici je uveden v rovnicích 5.1, 5.2 a 5.3.

$$Offset_0 = MI32Offset + RegArrOffset \tag{5.1}$$

$$Offset_n = Offset_{n-1} + |Regs_{n-1}| * 4 \tag{5.2}$$

$$Reg_{\{n,i\}} = Offset_n + 4 * i \tag{5.3}$$

Kapitola 6

Verifikace a dosažené výsledky

V této kapitole budou okomentovány dosažené výsledky a způsob verifikace funkcionality integrace stavové paměti. Z důvodu výkonnostního testování byla vytvořena testovací množina P4 programů s nejrůznějším zapojením stavových pamětí. Pro proces testování byla zvolena karta COMBO–100G2Q. Karta je osazena čipem FPGA Virtex–7 H580T umožňujícím běh komplexních programů.

Testování funkcionality bylo provedeno za použití verifikačního prostředí [8] vytvořeného sdružením CESNET a existující verifikace napsané v jazyce SystemVerilog. Simulačním nástrojem byl zvolen program ModelSim. Verifikace posílá desetitisíce transakcí skrze zařízení a porovnává přijaté výstupy s referenčním řešením behaviorálního modelu. Behaviorální model je softwarový program realizující funkcionalitu popsanou v P4 jazyce poskytovaný Konsorciem jazyka P4 [17]. Registry byly otestovány na statisících testovacích vektorech. Ve zvolených uživatelských akcích byla volána sekvence *write* a *read* operací přistupujících ke stejné paměťové buňce. V průběhu celého testu nebyla ani jednou narušena data obsažená v testovacích vektorech.

Testovací množina se skládá ze sedmi P4 programů, kde každý z nich používá stavovou paměť odlišným způsobem. Testovací množina byla vytvořena za účelem ověřit vygenerování VHDL zdrojových kódů pro různé případy užití a především pro demonstraci vlivu použití stavových objektů na výkonnost a hardwarové zdroje zařízení. Základem každého ze sedmi uvedených testovacích programů jsou tři tabulky. Nyní uvedu stručné charakteristiky jednotlivých programů:

reference.p4 Program slouží jako referenční řešení bez použití jakékoliv stavové paměti.

simple–reg Příklad deklaruje právě jedno statické registrové pole o třech paměťových buňkách mající velikost 8 bitů realizovanou v distribuované paměti.

reg–bram Jedná se o rozšíření předchozího programu simple–reg o dvě přímá registrová pole implementována v BRAM paměti přiřazena k tabulce s 512 záznamy. Uživatelská akce obsahuje 4 volání primitivních akcí pracující s registry.

simple–cnt Program je tvořen pouze jedním statickým čítačem s 5 položkami o šířce 32 bitů. V uživatelské akci je provedeno pouze jedno volání primitivní akce *count*.

cnt–direct–bram Program je tvořen pouze jedním přímým čítačem implementovaným v BRAM paměti a přiřazeným k tabulce s 512 záznamy. Šířka čítače je 11 bitů.

reg–cnt–together Příklad obsahuje jedno přímé registrové pole implementované v paměti BRAM a přiřazené k tabulce s 512 vstupy a šířkou paměťové buňky 16 bitů. Navíc jsou

ve stejné tabulce deklarovány dvě pole čítačů. První se skládá z 5 prvků a šířce 32 bitů. Druhé pole má 4 prvky o šířce 16 bitů. Během vykonávání uživatelské akce jsou vykonány dvě primitivní akce přistupující k registrům a čítačům.

cnt_dir–multi–reg Program byl vytvořen za účelem testování použití přímých čítačů a registrů. Dvě pole čítačů s pěti a čtyřmi prvky, jedno přímé pole čítačů a dvě přímá registrová pole jsou deklarována v tabulce s 512 záznamy. Další tabulka má statický čítač typu *packets_and_bytes* s 444 prvky a registrové pole o třech prvcích.

Tabulka 6.1 uvádí potřebné zdroje FPGA pro vytvoření zařízení a jeho maximální pracovní frekvenci. Cílová frekvence zařízení byla 200 MHz. Výsledky ukazují, že současná implementace efektivně šetří zdroje čipu a je schopna běžet na požadované cílové frekvenci.

Use case	**LUT([%])**	**BRAM([%])**	***f*** **[MHz]**
reference	125554(35)	219(23)	202.8
simple-reg	130671(36)	219(23)	200.1
reg-bram	131206(36)	220.5(23)	201.7
simple-cnt	131217(36)	219(23)	201.7
cnt-direct-bram	130386(36)	219.5(23)	204.8
reg-cnt-together	130930(36)	220.5(23)	203.2
cnt_dir-multi-reg	131277(36)	223.5(24)	202.9

Tabulka 6.1: Spotřebované zdroje pro jednotlivé programy testovací množiny

Výkonnost zařízení je ovlivněna zamykacím mechanismem, který pozastavuje práci zařízení po určitý počet hodinových taktů. Konečná propustnost zařízení je dána rovnicí 6.1, kde f značí pracovní frekvenci zařízení, L_i znamená latenci uživatelské akce a $Cnt_{dir} = max(Cdir_{reqCount} - 1, 0)$. $Cdir_{reqCount}$ označuje počet přístupů přímých čítačů v tabulce odkud byla latence L_i vybrána.

$$Thr = \frac{f}{max(L_1, L_2, L_3, ...L_n) + Cnt_{dir} + 1} \tag{6.1}$$

Latence uživatelské akce je určena použitými primitivními akcemi. Všechny primitivní akce, které nepracují se stavovou pamětí, jsou implementovány pomocí kombinačních logických obvodů. Proto uživatelské akce obsahující žádný či jeden přístup do stavové paměti (trvající jeden hodinový takt), mají latenci rovnu jednomu hodinovému taktu. V ostatních případech je výsledná latence uživatelské akce vypočtena podle následujících rovnic:

$$L_i = max(R_{op}, C_{op}, 1) \tag{6.2}$$

$$R_{op} = |R_{wr}| + |R_r| \tag{6.3}$$

$$C_{op} = |C_{count}| \tag{6.4}$$

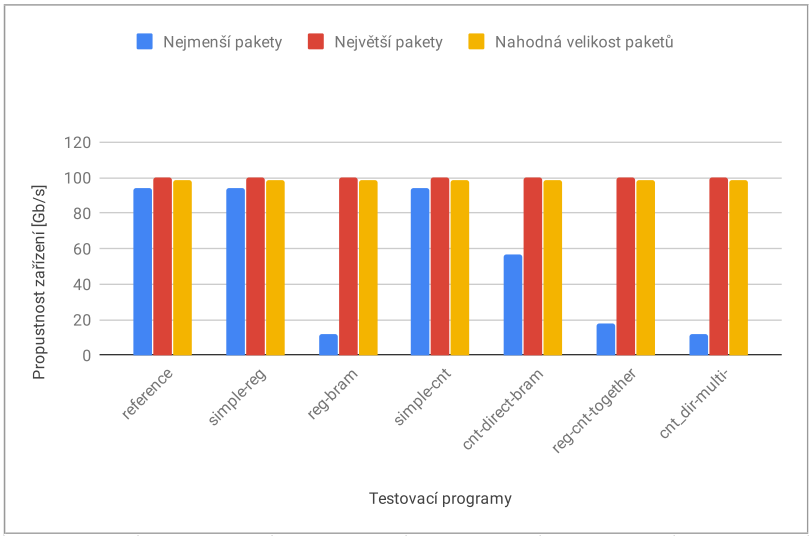
$|R_{wr}|, |R_r| \, and \, |C_{count}|$ značí počet hodinových taktů potřebných pro vykonání všech primitivních akcí *register_write*, *register_read* a *count* v dané uživatelské akci. Počet hodinových taktů potřebný pro přístup k dané paměti je uveden v tabulce 6.2.

Obrázek 6.1 ukazuje výsledky výkonnostního testování na lince s propustností 100 Gb/s. Každý program z testovací množiny byl testován pro tři specifické datové provozy. Konfigurace zařízení a generátoru síťového provozu byla provedena tak, aby cílila veškerý zpracovávaný provoz na akce používající stavové paměti. Proto obrázek 6.1 ukazuje chování

	Implementation	
Action	**Distribute RAM**	**BRAM**(output reg)
register_write	1	2(2)
register_read	1	1(2)
count	1	1(2)

Tabulka 6.2: Přístupová doba do jednotlivých stavových pamětí

zařízení pro extrémní situace síťového provozu, které se běžně v reálném světě nevyskytují. Pro každou extrémní situaci je vyčleněn jeden sloupec. Prvním vzorkem provozu je situace, kdy má každý přijatý paket velikost 64 bajtů. Díky této velikosti je datová linka schopna posílat velké množství paketů za vteřinu, což způsobuje zmenšení propustnosti zařízení. Propustnost zařízení je omezena pracovní frekvencí a možností zpracovávat jeden paket za hodinový takt. Pro zobrazení propustnosti zařízení při velikosti paketů 64 bajtů je vyčleněn modrý sloupec. Červený sloupec značí síťový provoz složený z paketů mající vždy velikost 1526 bajtů. V tomto případě není datová linka schopna posílat tak velké množství paketů jako v předchozím případě. Proto je výsledná propustnost zařízení dostačující pro zpracování všech paketů na lince bez ohledu na testovací program. Poslední žlutý sloupec je vyčleněn pro provoz složený z paketů nabývajících velikost od 64 do 1526 bajtů s rovnoměrným rozložením a průměrnou délkou paketu 919 bajtů.



Obrázek 6.1: Propustnost zařízení pro jednotlivé testovací programy

Výsledky ukazují, že propustnost zařízení pro nejdelší (červené) a náhodně generované (žluté) pakety je stejná pro všechny příklady uvedené v testovací množině a je dostačující i v případě použití stavových pamětí. Mechanismus zámku ovlivňuje výkonnost pouze v případě, kdy je provoz tvořen pakety minimální velikosti. Výsledky na nejmenších paketech potvrzují vztah pro výpočet výsledné propustnosti zařízení stanovený v rovnici 6.1.

Kapitola 7

Závěr

Cílem této práce bylo seznámení se s jazykem P4, existujícím kompilátorem vyvinutým v rámci skupiny Liberouter [11] a rozšířit stávající verzi kompilátoru o podporu práce s registry a čítači. Současně rozšířit i možnosti konfigurace přes knihovnu *libp4dev*. Všechny stanovené cíle se podařilo v rámci bakalářské práce splnit.

V počáteční fázi stáže u výzkumné skupiny Liberouter jsem se seznámil s jazykem P4 a jeho překladačem do HDL popisu pro FPGA, který je vyvíjen v rámci projektu NFV200 Technologické agentury České republiky. Vytvořil jsem implementaci P4 demo aplikace ukazující možnosti specifikace nového protokolu IPv7 právě pomocí popisu jazyka P4. Vytvořené demo bylo úspěšně prezentováno na mezinárodní konferenci IEEE FPL 2017 v Ghentu[5]. Po nastudování používaných technologií a vývojového prostředí byly navrženy a implementovány moduly pro realizaci práce s registry a čítači jazyka P4, které byly následně úspěšně integrovány do překladače. Pro zajištění konzistence dat byl systém rozšířen o mechanismus zamykání přístupů do uživatelských akcí. Současně bylo v průběhu vývoje navrženo a implementováno API pro práci s registry a čítači pro možnosti konfigurace jednotlivých modulů. Dále byl překladač rozšířen o generování informačního výpisu. Výpis poskytuje přehlednou informaci o způsobu implementace stavové paměti a předpokládané propustnosti zařízení při daném použití stavových pamětí s ilustrativním příkladem pro pracovní frekvenci 200 MHz na lince s datovou šířkou 100 Gb/s. Jednotlivé moduly stavových pamětí byly verifikovány v simulačním prostředí i na skutečném hardwarovém zařízení (akcelerační karta COMBO [12]). Výsledky propustnosti generované v informačním výpisu překladače byly ověřeny výkonnostním testováním na skutečném zařízení pomocí množiny P4 programů popsané v kapitole 6. Z měření vyplývá, že současné řešení registrů a čítačů je schopné dosahovat cílové propustnosti 100 Gb/s.

Realizačním výsledkem práce je implementace a následná integrace modulů registrů a čítačů do systému překladače jazyka P4 schopných dosahovat cílové propustnosti 100 Gb/s. Pro navýšení propustnosti zařízení pro programy obsahující větší počet požadavků na stavové paměti lze tyto požadavky rozdělit mezi více tabulek a přenášet dílčí mezivýsledky pomocí metadat programu. Případné snížení propustnosti zařízení odpovídá vztahu uvedenému v rovnici 6.1. Budoucí výzvou je rozšíření stavového zpracování o podporu využití externích pamětí s minimalizací vlivu na výslednou propustnost zařízení.

Celá práce vznikla ve spolupráci se sdružením CESNET [6] a byla úspěšně komercializována společností Netcope Technologies [1]. Na základě této práce byl vytvořen článek, který byl prezentován na Studentské Konferenci Inovací, Technologií a Vědy v IT s názvem Excel@FIT 2019, kde získal ocenění odborné komise za prakticky využitelný systém v oblasti akcelerace síťových funkcí [10].

Literatura

[1] Netcope Technologies. [Online; navštíveno 24.4.2019].
URL https://www.netcope.com/en

[2] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, Steve Licking: Packet Transactions: High-Level Programming for Line-Rate Switche.
URL https://pdfs.semanticscholar.org/9151/02d54fddae61591324982c584a3d0d87d58c.pdf

[3] Benáček, P.: *Ethernetový tester pro vysokorychlostní sítě*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2012.

[4] Benáček, P.: *Generation of High-Speed Network Device from High-Level Description*. Dizertační práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.
URL https://fit.cvut.cz/sites/default/files/PhDThesis-Benacek.pdf

[5] Benáček, P.; Puš, V.; Kořenek, J.; aj.: Line rate programmable packet processing in 100Gb networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, ISSN 1946-1488, s. 1–1, doi:10.23919/FPL.2017.8056835.

[6] CESNET: [Online; navštíveno 6.5.2019].
URL https://www.cesnet.cz

[7] Communications, S.: [Online; navštíveno 9.5.2019].
URL https://www.spirent.com/products/testcenter

[8] Iša, R.; Benáček, P.; Puš, V.: Verification of Generated RTL from P4 Source Code. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, ISSN 1092-1648, s. 444–445, doi:10.1109/ICNP.2018.00065.

[9] J. Heinanen and R. Guerin: A Single Rate Three Color Marker. RFC 2697, RFC Editor, September 1999.
URL https://tools.ietf.org/html/rfc2697

[10] Kohout, P.: Stateful Packet Processing In High-speed Network Devices Described In P4.
URL http://excel.fit.vutbr.cz/submissions/2019/007/7.pdf

[11] Liberouter: [Online; navštíveno 6.5.2019].
URL https://www.liberouter.org

[12] Liberouter: Síťová karta COMBO-100G2Q. [Online; navštíveno 10.5.2019].
URL https://www.liberouter.org/combo-100g2q/

[13] Netronome Systems, Inc.: Programming NFP with P4 and C.
URL
https://www.netronome.com/m/documents/WP_Programming_with_P4_and_C_.pdf

[14] Open Networking Foundation: OpenFlow Switch Specification version 1.5.1.
URL https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[15] P4 Language Consortium: P4.
URL https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[16] P4 Language Consortium: P4-HLIR.
URL https://github.com/p4lang/p4-hlir

[17] P4 Language Consortium: P4C-BEHAVIORAL.
URL https://github.com/p4lang/p4c-behavioral

[18] P4 Language Consortium: P4VHDL.
URL http://p4fpga.github.io/

[19] P4 Language Consortium: WEB.
URL http://p4.org/

[20] Puš, V.; Benáček, P.: Cesnet P4.
URL https://www.cesnet.cz/2016/01/jazyk-p4/

[21] Santiago del Rio, P. M.; Rossi, D.; Gringoli, F.; aj.: Wire-speed Statistical Classification of Network Traffic on Commodity Hardware. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1705-4, s. 65–72, doi:10.1145/2398776.2398784.
URL http://doi.acm.org/10.1145/2398776.2398784

Příloha A

IPv7 aplikace

```
header_type ipv7_t {
    fields {
        identification : 16;
        ttl : 8;
        protocolNumber : 16;
        nextHeader : 16;
        checksum : 24;
        srsAddr : 48;
        dstAddr : 48;
    }
}

header ethernet_t ethernet;
header ipv7_t ipv7;
header ipv4_t ipv4;

parser start {
    return parse_ethernet;
}
parser parse_ethernet {
    extract(ethernet);

    return select(ethernet.ethertype){
        0xbeef  :  p_ipv7;
        default :  ingress;
    }
}
parser p_ipv7 {
    extract(ipv7);

    return select(latest.protocolNumber){
        0xc0c0  : p_ipv4;
        default : ingress;
    }
}
```

```
parser p_ipv4 {
    extract(ipv4);
    return ingress;
}

action filter {
    modify_field(ethernet.ethertype, 0x800);
    remove_header(ipv7);
}

table filter {
    reads {
        ipv4.dstAddr   : exact;
    }

    actions {
        _pass;
        _drop;
    }
}

control ingress {
    apply(remove_ipv7);
    if(valid(ipv4))
        {
                apply(filter);
        }else
        {
                apply(drop_packet);
        }
}
```

Příloha B

Primitivní akce pro vytváření statistik v $P4_{14}$

count(counter_ref, index)

Primitivní akce pro inkrementaci hodnoty čítače.

- *counter_ref*: jméno pole s čítači.
- *index*: pozice čítače v poli.

Akce je povolené použít pouze v případě, kdy obsahem atributu *counter_ref* je pole čítačů deklarované jako *static* nebo *global*. Chování akce je dáno *typem* čítače a je popsáno v 2.2.3.

execute_meter(meter_ref, index, field)

Primitivní akce pro aktivace měřiče.

- *meter_ref*: jméno pole s měřiči.
- *index*: pozice měřiče v poli.
- *field*: políčko pro uložení výsledku.

Akce je povolené použít pouze v případě, kdy obsahem atributu *meter_ref* je pole měřičů deklarované jako *static* nebo *global*. Akce označí daný paket vypočítanou barvou na políčku *field*.

register_read(dest, register_ref, index)

Primitivní akce pro čtení hodnoty z registru.

- *dest*: jméno políčka pro uložení čtené hodnoty.
- *register_ref*: jméno registrového pole.
- *index*: pozice registru v registrovém poli. Pouze pokud není direct.

Daný registr určený parametry *register_ref* a *index* je zpřístupněn a jeho hodnota je uložena v *dest*. Pokud je registr typu direct, je hodnota parametru *index* ignorována a registr je určen na základě vstupu do tabulky.

register_write(register_ref, index, value)

Primitivní akce pro zápis hodnoty do registru.

- *register_ref*: jméno registrového pole.
- *index*: pozice registru v registrovém poli. Pouze pokud není direct.
- *value*: Políčko jehož hodnota bude uložena.

Hodnota z políčka daného parametrem *value* je zapsána do registru určeného parametry *register_ref* a *index*. Parametr *value* může být zadaná konstanta, nebo hodnota z parametru odpovídající vstupní akce. Pokud je registr typu direct, je hodnota parametru *index* ignorována a registr je určen na základě vstupu do tabulky.