VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PŘEKLADAČ JAZYKA P4$_{16}$ VYUŽÍVAJÍCÍ VYSOKOÚROVŇOVOU SYNTÉZU

COMPILER OF P4$_{16}$ USING HIGH-LEVEL SYNTHESIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **JAKUB NERUDA**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Neruda Jakub, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Překladač jazyka P4.16 využívající vysokoúrovňovou syntézu**
P4.16 Compiler Using High Level Synthesis
Kategorie: Počítačové sítě
Zadání:

1. Seznamte se s jazykem P4.16 a dostupnými nástroji pro tvorbu překladače tohoto jazyka.
2. Seznamte se s technikami a nástroji pro vysokoúrovňovou syntézu obvodů od společností Xilinx a Intel.
3. Navrhněte vhodný způsob realizace překladače jazyka P4.16. Zaměřte se část akcí a jejich realizaci s využitím vysokoúrovňové syntézy.
4. Proveďte implementaci navrženého překladače a jeho funkčnost ověřte na vybraných programech v P4.
5. Zhodnoťte dosažené výsledky a diskutujte další pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 25. října 2019

Abstrakt

Jazyk P4, určený pro programování funkcionality síťových prvků je v současnosti progresivním trendem na poli síťové administrace. Nicméně tento jazyk se stále vyvíjí a jeho poslední revize $P4_{16}$ výrazně změnila nejen možnosti jazyka a jeho syntax, ale i celý kompilátor. Sdružení CESNET podporuje vývoj P4, a proto i jeho se týká přechod na nový standard. Tato práce zkoumá možné problémy spojené s migrací, konkrétně překlad vysokoúrovňových uživatelských akcí do VHDL popisu s využitím vysokoúrovňové syntézy, zapojování speciálních extern objektů a podporu atomických sekcí. Text diskutuje možné způsoby zapojení HDL komponent, jakož i organizaci jejich paměťového prostoru pro runtime konfiguraci ze software. Taktéž je přiblížena architektura kompilátoru, s praktickými ukázkami realizace základních objektů pro překlad P4 do cílové architektury. Závěr práce demonstruje využití nástroje Vivado HLS pro optimalizaci C++ kódů za účelem co největšího výkonu výsledného obvodu.

Abstract

The P4 language is currently a hot topic in the field of network administration due to its capability to program the functionality of network devices. This language is still in development and its last revision $P4_{16}$ drastically changed not only the language features and syntax, but also the underlying compiler. The CESNET association supports the development of the P4 language and thus they also need to support the new standard. This work examines possible problems tied to migration, namely issues related to translation of high-level user-defined actions into VHDL description, with the help of High-level Synthesis (HLS), instantiation of so-called extern objects and the support of atomic sections. The text discusses possible ways of interconnecting the HDL components and organisation of their memory space in order to support configuration from software at runtime. The architecture of the p4c compiler is also described, complete with code examples implementing core classes participating in the compilation process. The last part of the work showcases the usage of Vivado HLS for optimizing C++ code in order to get maximum performance from the resulting firmware.

Klíčová slova

P4, kompilátor, SDN, HLS, vysokoúrovňová syntéza, optimalizace, Vivado

Keywords

P4, compiler, SDN, HLS, High-level Synthesis, optimizations, Vivado

Citace

NERUDA, Jakub. *Překladač jazyka $P4_{16}$ využívající vysokoúrovňovou syntézu.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Překladač jazyka $P4_{16}$ využívající vysokoúrovňovou syntézu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Tomáše Martínka, Ph.D. Další informace mi poskytl Pavel Benáček, Ph.D., Ing. Mário Kuka a Ing. Martin Žižka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Neruda
2. června 2020

Poděkování

Chtěl bych poděkovat všem svým kolegům za pevné nervy při práci se mnou.

Obsah

1 Úvod **2**

2 Koncepce a metodologie **4**
2.1 Jazyk P4 4
2.2 Vysokoúrovňová syntéza 11
2.3 Současný stav kompilátoru 15

3 Návrh **21**
3.1 Kritika současného stavu 21
3.2 Návrh realizace nové funkcionality 23
3.3 Konfigurace externů přes rozhraní MI32 26
3.4 Shrnutí 27

4 Implementace **28**
4.1 Práce s kompilátorem 28
4.2 Zapojení konfiguračního rozhraní MI32 do bloku akcí 31
4.3 Implementace externu Register 32
4.4 DeviceTree 35
4.5 Shrnutí 36

5 Experimenty a testování **37**
5.1 Experimenty 37
5.2 Testování 43
5.3 Shrnutí 44

6 Závěr **46**

Literatura **47**

Kapitola 1

Úvod

Jedním z velkých pojmů současnosti na poli konfigurace sítí je také programovací jazyk P4. Tento funkční i duchovní nástupce technologie OpenFlow umožňuje lépe programovat parser paketů a akce aplikované na pakety. Je tudíž možné mnohem komplexněji nakonfigurovat konkrétní prvek tak, aby plnil svou roli v síťové architektuře. Síťový administrátor není omezen předkonfigurovanou logikou čipu danou výrobcem, ale pouze hardwarem síťového zařízení, a tudíž může podstatně efektivněji pracovat se zdroji, které má k dispozici.

Výzkumné sdružení CESNET je jednou z organizací, které v P4 spatřují budoucnost pro síťový hardware a správu sítí. Proto ve spolupráci s partnerskou firmou Netcope Technologies[1] vyvíjí vysokorychlostní síťové karty osazené FPGA čipy, jejichž činnost je možné popsat právě v jazyce P4 a následně kompilovat[2] do VHDL. Z VHDL jde následně snadno vygenerovat bitstream, pomocí kterého lze rekonfigurovat FPGA čip a díky tomu se efektivně dá chování těchto karet programovat pomocí jazyka P4.

Tato přidaná úroveň abstrakce ovšem zvyšuje nároky na vývojáře, kteří musí neustále přicházet se způsoby, jak obecné koncepty jazyka P4 nejen přetvořit do hardwarového popisu, ale navíc jak dosáhnout dostatečně výkonného popisu hardwaru, který nebude mít negativní dopad na propustnost karet, od kterých se vyžadují i rychlosti dosahující dvou set gigabit za sekundu.

Pro zjednodušení a standardizaci překladu vyvíjí skupina p4lang (která tvoří i standard jazyka) front a mid-end komponenty kompilátoru. Na tyto komponenty pak vývojáři mohou napojit své back-end komponenty generující hardwarově agnostický výstup. Tento formát kompilace byl zaveden spolu s novou revizí standardu jazyka $P4_{16}$ a pro CESNET představuje příležitost, jak výrazně zdokonalit proces generování VHDL popisu. Mezi to patří i využití modernějších nástrojů pro popis chování hardware, jako je například HLS — High-level Synthesis (česky vysokoúrovňová syntéza), což mimo jiné výrazně usnadní syntézu uživatelských $P4_{16}$ akcí.

Cílem této práce je seznámit se s back-end komponentou pro $P4_{16}$ kompilátor, která je vyvíjena sdružením CESNET, a s procesem překladu kódu uživatelských P4 akcí, identifikovat potenciální kritické body a navrhnout jejich řešení, které nebude mít negativní vliv na propustnost syntetizovaných obvodů. Dále je cílem naimplementovat podporu překladu programů využívajících externí objekty a atomické sekce.

[1] Pokud text práce zmiňuje sdružení CESNET nebo firmu Netcope Technologies bez explicitního jmenování obou entit, má vždy na mysli obě entity, uvedena je většinou ta, pro kterou je konkrétní téma relevantnější.

[2] Čistě technicky je korektní termín transpilace, nicméně autoři jazyka označují proces jako kompilaci.

V kapitole Koncepce a metodologie (2) text přibližuje pojmy P4, flow tabulka, kompilátor p4c a HLS. Dále diskutuje současný stav kompilátoru jazyka P4, vyvíjeného sdružením CESNET, a také konstrukty jazyka $P4_{16}$, které jsou problematické při tvorbě výstupního VHDL popisu. Kapitola Návrh (3) diskutuje nedostatky současné implementace a navrhuje možná řešení podpory atomických sekcí, extern objektů a jejich paměťového prostoru pro runtime konfiguraci. Kapitoly Implementace (4) a Experimenty (5) se zaměřují na praktickou realizaci variant vybraných kapitolou Návrh a jejich optimalizaci.

Kapitola 2

Koncepce a metodologie

Tato kapitola vysvětluje pojmy a širší kontext relevantní pro řešení práce. Kontext zahrnuje přiblížení jazyka P4 a jeho verzí a taktéž kompilátor p4c použitý pro převod P4 do VHDL. Dále je přiblížen syntézní nástroj Vivado HLS, jehož využití je jedním z hlavních bodů řešení této práce. Popsán je také současný stav implementace P4 sdružením CESNET, která tvoří základ pro realizaci této práce.

2.1 Jazyk P4

P4 (jméno vychází z názvu článku Programming Protocol-Independent Packet Processors [3]), je jazyk určený k programování generického síťového prvku tak, aby byl tento prvek schopný vykonávat konkrétní činnost jako třeba routing, switching, de/enkapsulace protokolů, či filtrace. Takto naprogramovaný síťový prvek je navíc ještě dále konfigurovatelný skrze vkládání pravidel do programátorem definovaných tabulek, přičemž každé pravidlo je vázáno na konkrétní hodnotu či rozsah vyhledávacího klíče, kterým jsou typicky položky z hlavičky paketu.

Specifikace a koncept tohoto programovacího jazyka je evolucí standardu OpenFlow [10], a hlavní motivací pro nahrazení tohoto standardu bylo především zbavit se závislosti na standardizačních komisích a výrobcích síťového hardware, neboť tyto výrazně komplikovaly práci výzkumným skupinám, jejichž zájmem bylo experimentovat s novými síťovými protokoly na produkční infrastruktuře.

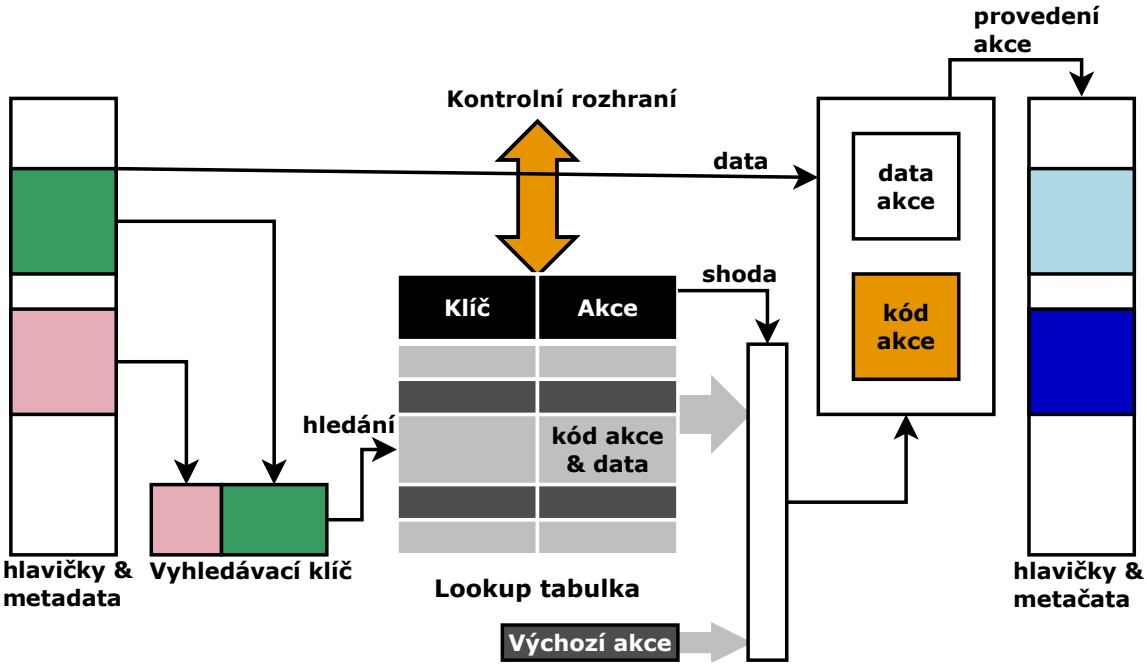
V produkční sféře je představa o využití jazyka poněkud odlišná. Hlavní případy využití této technologie tkví v dynamické konfiguraci vyvažování zátěže, rekonfiguraci sítě za běhu nebo také dynamickém šifrování provozu. Firmy nejeví zájem o experimentování s novými protokoly, preferují spíše možnost vysokého a rychlého škálování nad velkými tabulkami. Velkou výhodou P4 je i možnost změny požadavků bez nutnosti obměnit hardware. S tím se pojí i typ hardware, na které je možné takové řešení nasazovat. Zatímco OpenFlow cílilo v podstatě výhradně na oblast přepínačů, P4 umožňuje flexibilní popis i pro jiné prvky, například SmartNIC zařízení.

Match-Action tabulka

Match-Action (M/A) tabulky jsou základním stavebním kamenem jazyka P4 a jsou pokračováním konceptu zavedeného již v OpenFlow pod názvem flow tabulka. M/A tabulka je datová struktura sestávající z pravidel ve formátu klíč-akce, přičemž klíč je seznam hodnot vybraných polí z hlavičky paketu a akce je operace, která se na paket aplikuje, pokud se

vybraná pole v paketu shodují s hodnotami klíče (proto Match-Action, česky Shoda-Akce). Taktéž existuje možnost definovat výchozí akci, která se použije v případě, kdy žádné pravidlo nenašlo shodu (viz obrázek 2.1).

Každá tabulka definuje, jaká pole z hlaviček paketů budou využita klíčem, a taktéž jaký algoritmus pro nalezení shody[1] bude pro každé pole použit. Kupříkladu, pokud chceme zahodit veškeré pakety přicházející ze sítě 8.8/16 jdoucí na `cílový TCP port` 42, pak klíč používá pole `zdrojová adresa` z hlavičky `IPv4` s algoritmem LPM (longest prefix match) a pole `cílový port` z hlavičky `TCP` s algoritmem přesné shody.



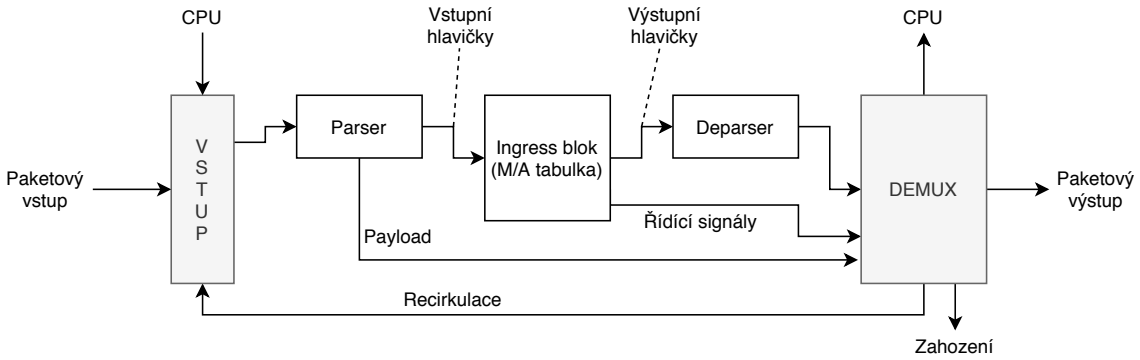
Obrázek 2.1: Schéma systému shoda+akce [15]

Síťový administrátor, jakmile jednou naprogramuje síťový prvek patřičným P4 programem, musí ještě nakonfigurovat obsah M/A tabulek skrze vložení příslušných pravidel ve formátu klíč-akce. Konkrétní forma konfigurace je platformě závislá a vyjma formální specifikace (P4 Runtime) napojení na existují SDN kontroléry neexistuje unifikovaný přístup k této činnosti [13].

Struktura P4 programu

Obrázek 2.2 ukazuje blokový diagram velice jednoduchého P4 programu. Bílé bloky jsou definovatelné pomocí P4, datové cesty a šedé bloky musejí být implementované cílovou architekturou. Řídící signály, nastavované skrze akce M/A tabulky, slouží ke zvolení správného chování v DEMUX bloku. Ten by měl být schopný paket zahodit, recirkulovat či třeba duplikovat do software.

[1] V praxi se často používá pojem search engine.



Obrázek 2.2: Blokový diagram jednoduchého P4 programu

P4 program se vždy skládá z Parser bloku, který interpretuje pakety na vstupu a extrahuje z nich hlavičky, jednoho či více kontrolních bloků (zde je pouze jeden – Ingress blok), kde jsou instanciované všechny M/A tabulky a externy a nakonec Deparser blok, který skládá aktualizované hlavičky z interních struktur zpět do validní hlavičky paketu. Jazyk P4 umožňuje definovat modely, které specifikují dodatečné povinné kontrolní bloky například pro ověření a aktualizaci kontrolních součtů paketů [15].

Kód 2.1 ukazuje, jak může vypadat minimální program v $P4_{16}$, jehož struktura odpovídá obrázku 2.2. Jediným rozdílem je absence řídících signálů. Uvedený program lze použít pouze pro pozměnění hodnot vybraných hlaviček paketu, nikoliv k jeho přeposílání, recirkulaci, či zahazování.

Uvedený program se skládá z několika základních sekcí:

- Zahnutí jiných modulů skrze direktivu `#include`. Soubor `core.p4` obsahuje standardizované komponenty jazyka. Programátor ale může chtít zahrnout i konkrétní modely (v1model) či definice rozhraní extern objektů (PSA [14]).
- Definice hlaviček podporovaných protokolů – jde o výčet polí, v pořadí, v jakém se v protokolu vyskytují, přičemž každé pole je pojmenované, abychom s ním mohli dále pracovat a každé pole má jasně specifikovanou bitovou šířku.
- Deklarace struktur – struktura `headers` v příkladu je místo, kam se pro nás důležité části paketu vyparsují, kde je můžeme měnit a na konci programu zase zapsat do paketu.
- Definice parseru paketů – specifikuje, jakým způsobem jdou protokoly v paketu za sebou, a parsuje jejich hodnoty do připravených proměnných. Podporuje i větvení na základě hodnoty nějakého již vyparsovaného pole (například pole protocol z ethernetové hlavičky). Jde de-facto o kontrolní blok s alternativní syntaxí, která je relevantní pro parsování.
- Definice kontrolních bloků – každý kontrolní blok operuje nad dříve definovanými proměnnými, má vlastní sadu tabulek a akcí, a v `apply` bloku může specifikovat pořadí a podmínky, jak se mají tabulky aplikovat na vstupní pakety. Můžeme vidět, že deparser, který zapisuje hodnoty proměnných do paketu je obyčejný kontrolní blok, který v parametrech obsahuje `packet_out` parametr.

```
#include <core.p4> // standard language components

header data_t {
    bit<32> f1;
    bit<16> f2;
}

struct headers {
    data_t data;
}

parser Parser(packet_in pckt, out headers hdr) {
    state start { transition parse_data; }

    state parse_data {
        pckt.extract(hdr.data);
        transition accept;
    }
}

control Ingress(inout headers hdr) {
    action a1(in bit<16> value) { hdr.data.f2 = value; }

    table t1 {
        key = { hdr.data.f1 : exact; }
        actions = { a1; }
    }

    apply { t1.apply(); }
}

control Deparser(packet_out pckt, in headers hdr) { pckt.emit(hdr.data); }

Example (
    Parser(),
    Ingress(),
    Deparser()
) main;
```

Kód 2.1: Minimální $P4_{16}$ program

- Definice akcí – uživatelské operace, které mohou být aplikovány na zpracovávaný paket. Jejich účelem je obvykle modifikovat data paketu, v příkladu se konkrétně nahrazuje hodnota pole `f2` hodnotou `value`.
- Definice hlavního programu – specifikuje pořadí provedení kontrolních bloků, mezi kterými je stav sdílen a předáván skrze definované proměnné. První kontrolní blok by měl být vždy parser a poslední by měl být vždy deparser.

Ve formálnější rovině jsou akce, tabulky, kontrolní bloky i celý P4 program funkce, které transformují vstupní vektor bitů (příchozí paket) na výstupní vektor bitů (výstupní paket).

Oproti OpenFlow (s jeho velice standardizovaným chováním) je administrátor u P4 pravidel limitován pouze provozovaným P4 programem. Pokud si vezmeme jako příklad program uvedený v kódu 2.1, pro tabulku `t1` je možné vkládat pouze pravidla využívající pole `data.f1` s algoritmem přesné shody a akci `a1` s jedním povinným parametrem délky 16 bitů. Pokud bychom chtěli mít kupříkladu možnost paket zahodit, museli bychom tabulce `t1` do sekce `actions` přidat akci `drop`, program nově přeložit a nahrát do síťového prvku.

$\mathbf{P4_{16}}$

$P4_{16}$ je označení pro nejnovější revizi jazyka P4 z roku 2018; formálně je tato revize označena jako verze 1.1.0 [15]. $P4_{16}$ přináší oproti předchozí revizi $P4_{14}$ [16] řadu změn, přičemž pro tuto práci jsou významné pouze změny týkající se definice uživatelských akcí a s nimi souvisejících komponent. $P4_{14}$ umožňovalo definovat pouze takové uživatelské funkce, které ve svém těle sekvenčně volaly buď nějakou ze standardem definovaných primitivních akcí, nebo jinou uživatelskou akci (nerekurzivně).

$P4_{16}$ umožňuje v definici funkcí navíc používat následující:

- Deklarace atomických sekcí – blok kódu, který musí být vykonán, jako by šlo o jediný atomický příkaz.
- Výrazy složené z aritmetických a logických operací – $P4_{14}$ vyžadovalo použití primitivních funkcí a okamžité uložení mezivýsledku do hlavičky paketu.
- Podmíněné sekce – pouze jednoduché if-else bloky.
- Volání metod obecných extern objektů – registry, čítače, externí paměti.
- Return/exit – první jmenované zastaví vykonávání aktuálního bloku (akce/kontrolní blok), druhé zastaví vykonávání programu nad daným paketem.

$P4_{16}$ stále neumožňuje provádět rekurzi, vytvářet cykly, deklarovat lokální proměnné, či deklarovat switch sekce. Je vhodné zmínit, že jazyk $P4_{16}$ stále ještě prochází rapidním vývojem i na úrovni základní specifikace a je možné se tak setkat jak s novější verzí, uvedené v kódu 2.1, tak i se starší verzí, označovanou jako V1 model. Nejzásadnější rozdíl je ve skladbě kontrolních bloků – V1 model vyžaduje definici kontrolních bloků pro výpočet kontrolních součtů nad hlavičkami paketů. V1 model také používá vlastní hlavičkový soubor, obsahující i definice základních externích objektů (tzv. externů, viz dále). Novější verze standardu jazyka tyto definice vyčleňuje do separátního standardu PSA – Portable Switch Architecture [14].

Kód 2.2 ukazuje kód pro deklaraci a instanciaci jednoduchého extern objektu. Takový objekt se buď instanciuje globálně a do kontrolního bloku se předává přes parametr, nebo lokálně uvnitř konkrétního bloku.

```
extern Counter {
    Counter();
    void reset();
    void inc();
    bit<16> get();
}

Counter() cnt;
```

Kód 2.2: Deklarace a instanciace extern objektu

Deklarace extern objektu je víceméně ekvivalentní k deklaraci abstraktní třídy v běžném OO jazyce. P4 program potřebuje znát pouze rozhraní objektu, konkrétní implementace je na poskytovateli patřičného modulu. Takto nadefinovaný objekt jde pak jednoduše použít v kódu akce, jak ukazuje kód 2.3.

```
action a2(in bit<16> control) {
    if (user_metadata.control == control) {
        cnt.reset();
        user_metadata.egress_port = 0;
    }
    else {
        @atomic {
            cnt.inc();
            user_metadata.egress_port = cnt.get() % 8;
        }
    }
}
```

Kód 2.3: Kód akce s přístupem k extern objektu a atomickou sekcí

Kód 2.3 demonstruje program komplexnější akce, která by mohla realizovat velice jednoduchý vyvažovač zátěže – každý příchozí paket zvedne hodnotu čítače a na základě aktualizované hodnoty se přesměruje na konkrétní výstupní port. Vhodně zvolená kontrolní hodnota umožňuje čítač vyresetovat. Jak si můžeme všimnout, uvedený kód demonstruje tři věci – přístup k externím objektům, podmíněnou sekci a atomickou sekci.

Podmínky jsou syntakticky i sémanticky shodné s jazykem C. Jak již bylo řečeno dříve, extern objekt je adresovatelný skrze pojmenovanou proměnnou a definované rozhraní s neznámou implementací. Specifikace $P4_{16}$ vyžaduje, aby se každá metoda extern objektu chovala atomicky.

Ukázkový kód vyžaduje, aby se do položky `egress_port` uložila aktuální hodnota čítače. První větev podmínky ví, že čítač po resetování (které je atomické) obsahuje nulu, takže ji rovnou přiřadí do patřičné proměnné. Druhá větev podmínky netuší, jaká je aktuální hodnota čítače, a protože mezi voláním inkrementace čítače a získáním jeho hodnoty by mohlo teoreticky dojít k aktualizaci čítače z jiného zdroje (další paket, zápis z kontrolního software, jiná akce), musíme inkriminované dva řádky kódu obalit anotací `@atomic`, která zajistí, že se budou vždy vykonávat tak, jako by šlo o jediný příkaz.

Způsob, jakým bude atomická sekce naimplementována, je definován nikoliv standardem, nýbrž možnostmi cílové platformy.

p4c

Zkratka p4c je pojmenování oficiálního repozitáře uskupení p4lang, v rámci kterého je implementována platformě nezávislá část kompilátoru. Sestavovací systém je navržen takovým způsobem, aby do něj šla snadno přidat implementace platformě závislých komponent, tedy komponent, které emitují výsledný kód pro cílovou platformu.

Struktura celého kompilátoru je rozdělena do tří částí:

- Frontend – načítá vstupní zdrojové kódy a transformuje je do interní reprezentace. Podporuje jak $P4_{16}$, tak i $P4_{14}$ programy.
- Midend – provádí transformace a optimalizace nad načteným kódem. V této fázi se vytvářejí objekty `TypeMap` a `ReferenceMap` sloužící pro pozdější získávání definicí a typových informací o objektech v kódu.
- Backend – implementuje platformě závislé komponenty. Tyto komponenty by měly emitovat výstup potřebný pro naprogramování konkrétní platformy. Oficiální repozitář obsahuje tři demonstrační backendy:
  - p4test – backend vytvořený pro testovací účely
  - ebpf – backend pro extended Berkeley Packet Filters[2]
  - graphs – backend pro renderování grafů z P4 programů
  - bmv2 – backend pro simulační softwarový switch zvaný behavioral model v2[3]

Jednotlivé komponenty kompilátoru si mezi sebou předávají IR (Intermediate Representation), který je za každých okolností zpětně přeložitelný do P4 zdrojového kódu, který by měl být funkčně ekvivalentní ke vstupnímu kódu. Zároveň si IR uchovává informace o pozici jednotlivých svých uzlů v rámci vstupního zdrojového programu. Obě rozhodnutí byla učiněna s cílem zjednodušit ladění algoritmů a reportování chyb.

Algoritmy v rámci jednotlivých komponent jsou implementovány pomocí návrhového vzoru Návštěvník (Visitor) [8]. Tento návrhový vzor je výhodný v tom, že odděluje algoritmus od dat, a pokud přidáme do systému nový datový typ, musíme upravit pouze ty třídy (algoritmy), které nový datový typ chtějí používat, zatímco ostatní třídy zůstanou neovlivněny. Vzor Návštěvník proto dodržuje Open-Closed Principle (OCP), tedy že třída má být otevřena vůči rozšíření, ale zavřená vůči změnám [11].

V případě P4 kompilátoru umožňuje Návštěvník snadno rozšířit standard jazyka $P4_{16}$, a tudíž i funkčnost kompilátoru například o novou aritmetickou operaci (tedy nový datový typ IR uzlu), a pouze třídy, které implementují algoritmy nad aritmetickými uzly se musí změnit tak, že přidají novou metodu zpracovávající uzel nového IR typu.

Jednotlivé komponenty kompilátoru disponují posloupností objektů odvozených od základních tříd pro inspekci a transformaci IR programu, které se v několika průchodech aplikují na program a upravují jeho strukturu. Mezi tyto transformace patří obvyklé operace jako eliminace mrtvého kódu, propagace konstant či transformace jednoduchých if-else podmínek do ternárních operátorů.

Kód 2.4 ukazuje implementaci modelového minimálního algoritmu, který se bude aplikovat jen na uzly typu `Type_Specialized` (modelový příklad). Tento algoritmus pouze vypíše na standardní výstup textovou reprezentaci podstromu daného uzlu a pak skončí,

[2] http://man7.org/linux/man-pages/man2/bpf.2.html
[3] https://github.com/p4lang/behavioral-model

```
class MyAlgorithm : public IR::Inspector {
public:
    bool preorder(const IR::Type_Specialized *node) {
        ::dump(node);
        return false;
    }
};

// pouziti algoritmu. Promenna program obsahuje IR celeho programu
MyAlgorithm alg();
program->apply(alg);
```

Kód 2.4: Příklad minimálního algoritmu nad IR

přičemž podstrom uzlu node už nebude dále prohledáván. Aplikace algoritmu, s využitím návrhového vzoru Návštěvník, funguje tak, že nainstanciujeme objekt implementující algoritmus a předáme ho zkoumanému uzlu jako parametr metody `apply`. Tato metoda projde celým grafem (včetně zdrojového uzlu), a pokud nalezne kompatibilní uzel, algoritmus se nad ním spustí.

Všimněme si binární návratové hodnoty funkce `preorder`. Vrácením hodnoty `true` programátor specifikuje, že ačkoliv se algoritmus úspěšně aplikoval na současně procházený uzel, má se procházet i podstromem daného uzlu. Opačná hodnota by toto další procházení zastavila.

Způsob procházení navíc můžeme ovlivnit použitím jiné metody, než je `preorder`. Programátor může algoritmus napsat s využitím metod `inorder` a `postorder`, u kterých je ovšem alespoň část podstromu uzlu prozkoumaná ještě před tím, než se aplikuje konkrétní metoda s algoritmem. Tyto dvě metody jsou tím pádem v praxi použitelné jenom málokdy.

2.2 Vysokoúrovňová syntéza

Vysokoúrovňová syntéza (HLS – High-level synthesis) představuje v současnosti rostoucí trend na poli popisu hardware. V HLS se pro popis hardware používají vysokoúrovňové jazyky, které se tradičně využívají pro programování software jako například C nebo C++. Tento vysokoúrovňový popis se při syntéze transformuje do jazyků jako VHDL či Verilog, které se následně dají překládat do RTL (Resistor-transistor logic) skrze existující syntézní systémy.

Použití HLS výrazně zvyšuje produktivitu práce vývojářů, neboť umožňuje odladit a otestovat vyvíjenou komponentu v čistě softwarovém prostředí a až poté optimalizovat datové šířky použitých proměnných či použité komunikační rozhraní, přičemž tyto optimalizace se stále dějí na úrovni zvoleného programovacího jazyka a stále jsou verifikovatelné v softwarových testech. Omezení mající dopad na výslednou latenci a inicializační interval obvodu nicméně stále mohou způsobit nechtěnou změnu logiky a musejí tedy být znovu ověřeny ve verifikacích.

Je pochopitelné, že tento přístup zvyšuje nároky na syntézní nástroj, který musí vysokoúrovňový popis transformovat do hardwarového popisu a následně do RTL. Tyto transformace se dějí v pěti základních fázích [4]:

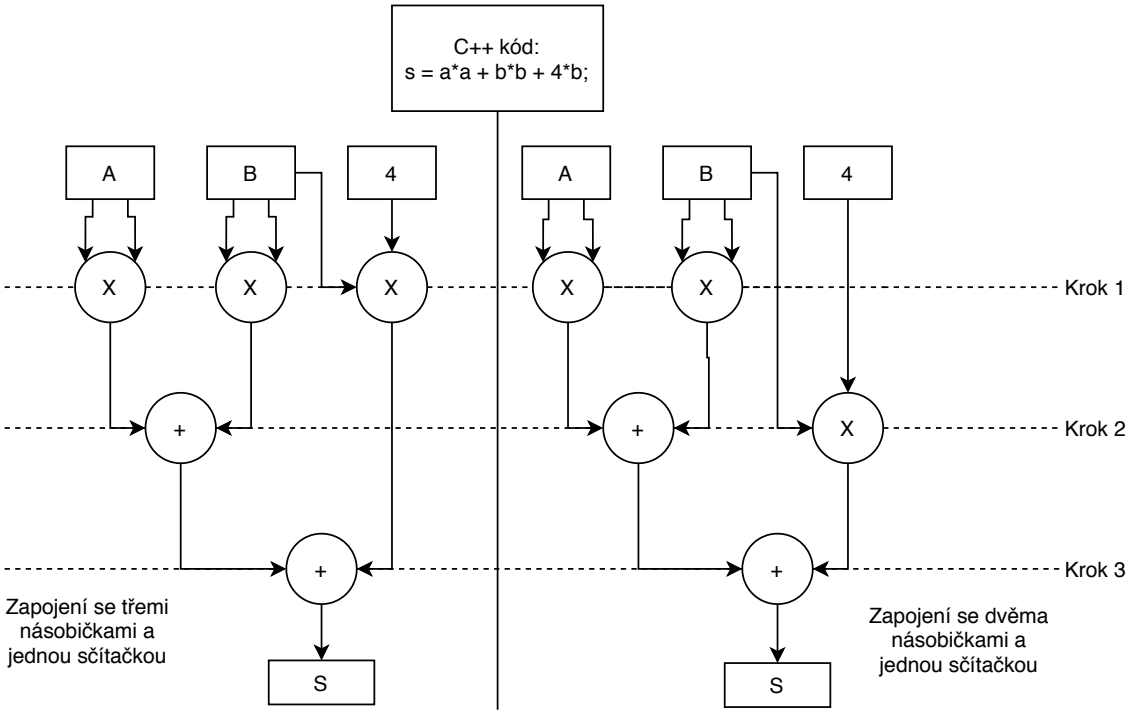
- Kompilace a modelování – v této fázi je použit nástroj g++ k překladu C/C++ zdrojových kódů do vnitřní behaviorální reprezentace výsledného obvodu. Behaviorální reprezentace se obvykle znázorňuje jako orientovaný graf (control flow graf), kde hrany jsou datové závislosti a uzly jsou konkrétní operace.

- Alokace – během alokace dochází k rezervování komponent přítomných na výsledném čipu, které budou využity pro výpočet nad grafem – například registry, násobičky, sčítačky. Obrázek 2.3 ukazuje, jaký vliv může mít různá alokace zdrojů na výslednou podobu grafu.

- Plánování – při plánování se berou v potaz pouze dostupné alokované zdroje a behaviorální popis výsledného obvodu. Cílem plánování je vytvořit takový graf odpovídající behaviorálnímu popisu, který bude mít co nejlepší časování. Jak ukazuje obrázek 2.3, alokované zdroje mají velký vliv na to, kdy se jednotlivé paralelizovatelné operace mohou skutečně dít. Pokud bychom kupříkladu alokovali pouze jednu násobičku, násobení už by nebylo možné jakkoli paralelizovat a výsledná latence obvodu by byla 4. Stejně tak inicializační interval by bylo nutné posunout na 2, anebo logiku vhodně proregistrovat. Naopak přidání další sčítačky by nijak nepomohlo, protože graf jasně ukazuje datovou závislost mezi sčítacími operacemi, a tudíž je nejde paralelizovat.

- Přiřazení – přiřazení spolu s alokací umožňuje mapování instrukcí a proměnných behaviorálního popisu na hardwarové zdroje.

- Generování RTL – z naplánovaného grafu behaviorálního popisu je vytvořen RTL popis, nad kterým je možné provést klasickou syntézu, a to včetně typických optimalizačních kroků, jako je třeba deduplikace podgrafů AIG (And-inverter graph) a výsledné mapování do cílové technologie.



Obrázek 2.3: Control flow graf znázorňující různé realizace téže logiky při jinak alokovaných zdrojích

Různé syntézní nástroje mohou drobně měnit pořadí některých operací, nebo používají jiné plánovací algoritmy za účelem optimalizace konkrétních typů algoritmů a komponent. Pro tuto práci bude nadále relevantní pouze prostředí nástroje Vivado HLS od společnosti Xilinx, jelikož je používáno i sdružením CESNET[4].

Pipelining

Z optimalizací, které je možné aplikovat na vybrané kusy kódu v HLS, patří pipelining mezi ty používanější. V principu jde o to vzít blok logiky, jehož latence při dané frekvenci hodin je delší, než jeden takt. Tento blok se následně musí duplikovat/proregistrovat takovým způsobem, aby se do něj dala posílat data ideálně každý takt a aby po uplynutí úvodního zpoždění (latence), blok na výstupu generoval data rovněž každý takt.

Tento přístup je známý i z běžných procesorů, kde pipelining probíhá na každém vláknu procesoru při zpracovávání instrukcí. U normálního procesoru je ovšem efektivita pipeliningu značně snižována podmínkami a skoky do podprogramů, neboť hardware procesoru dopředu nezná, co bude program vykonávat. V případě HLS bloku k těmto poklesům efektivity nedochází. Podobně jako u procesorového pipeliningu i u HLS je ovšem potřeba řešit klasické paměťové konflikty, které vyvstávají z paralelního přístupu do stejného paměťového prostoru. Jmenovitě se jedná o konflikty:

- Čtení po zápisu (Read-after-write – RAW), taktéž zvané „pravá závislost“. Tento případ nastává tehdy, když instrukce potřebuje číst místo v paměti, které je výsledkem

[4]V práci je diskutován i nástroj Quartus HLS od společnosti Intel. I ten je relevantní pro projekt kompilátoru, avšak v průběhu tvorby práce nebyla možnost provádět na něm experimenty, a tudíž navrhovaná řešení vždy berou v potaz primárně Vivado HLS.

předchozího zápisu. V tomto případě je nutné čtení zpozdit, jinak by došlo ke čtení staré hodnoty paměti.

- Zápis po čtení (Write-after-read – WAR), taktéž nazývané anti-závislost. Případ nastává tehdy, když instrukce potřebuje zapsat hodnotu do paměti, která se předtím čte.
- Zápis po zápisu (Write-after-write – WAW). Nastává tehdy, když postupné zápisy do paměti musejí být provedeny v konkrétním pořadí, jinak by se aplikace mohla dostat do nekonzistentního stavu. Uvažujme následující příklad:
  - `I1: R1 = R2 * R3`
  - `I2: R1 = R4 + R5`
  - V tomto případě musí být zajištěno, aby se do R1 zapsal výsledek I2 až po zapsání I1 (přestože vykonávání I2 může trvat kratší dobu). Zároveň je potřeba zkontrolovat, zda výsledek instrukce I1 je nějak užitečný (kupříkladu může jít o zápis do registru na sdílené sběrnici), jinak bychom mohli I1 kompletně vyoptimalizovat pryč.

Výše uvedené konflikty jsou nejčastěji relevantní tehdy, když pipelinujeme výpočty nad sdílenou pamětí. V takovém případě musí HLS nástroj upravit inicializační interval a latenci řetězeného bloku, aby konflikty vyřešil. Nicméně, HLS mívá tendence datové závislosti přeceňovat, aby se zabránilo nechtěným chybám v logice.

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
    for (i = 0; i < INPUT_SIZE; i++) {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if (old == val) acc++;
        else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}
```

Kód 2.5: Výpočet histogramu v HLS s potenciální datovou závislostí

Kód 2.2 demonstruje případ, ve kterém je výpočet histogramu, jehož logika využívá pipelining. Nástroj Vivado HLS detekuje, že vnitřní cyklus přistupuje do stejné paměti `hist` na indexech `old` a `val`. Přestože podmínka, kontrolující přístup do paměti jasně specifikuje, že hodnoty `old` a `val` jsou různé (a tudíž mezi nimi neexistuje žádná datová závislost), nástroj to nedokáže detekovat a naplánuje čtecí a zápisové operace do alternujících taktů, čímž docílí inicializačního intervalu 2. Uživatel je ovšem na toto upozorněn v průběhu kompilace a může tyto falešné závislosti eliminovat skrze direktivu DEPENDENCE [17].

Není neobvyklé, aby logika, kterou chceme řetězit, obsahovala několik podmíněných větví, z nichž každá sama o sobě má jinou latenci a inicializační interval, a které je i možné plně paralelizovat bez datových závislostí. Přestože by tyto větve logiky bylo často možné plně paralelizovat a řetězit nezávisle na sobě, Vivado HLS dokáže pipelining aplikovat pouze na úrovni HLS bloku jako celku. Tím pádem bude mít výsledný obvod vždy tu nejhorší latenci a nejhorší inicializační interval ze všech dostupných větví logiky. Naproti tomu nástroj Quartus HLS má koncept interní paralelizace, který je nazván System of Tasks. Tento koncept v podstatě umožňuje definovat vícevláknové programy, kde jednotlivé větve pracují plně nezávisle a je tak možné na jednotlivých sekcích dosáhnout výrazně většího výkonu, pokud si to aplikace žádá [7].

Stream

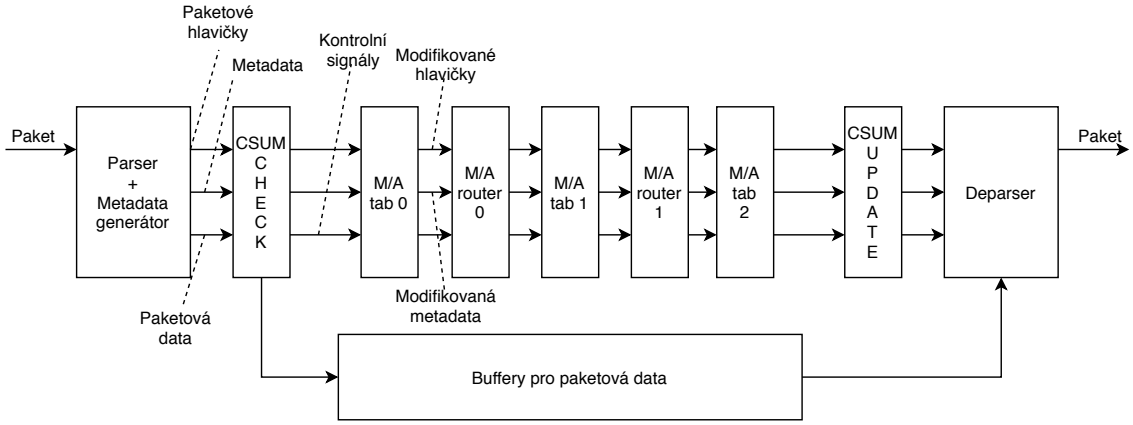
Stream je speciální třída dostupná pouze pro HLS kódy psané v C++, která poskytuje příhodné rozhraní pro komunikaci mezi komponentami. Streamy jsou nejčastěji využívány jako rozhraní HLS bloku, jelikož je možné je nakonfigurovat pro tři nejčastěji používané komunikační metody (FIFO, handshake a AXI) a jelikož vedou na jednodušší logiku, než kdyby programátor použil v rozhraní HLS bloku ukazatel do paměti (což je taktéž možné). Streamy je možné použít i pro realizaci interní komunikace v HLS bloku, v takovém případě jsou vždy implementovány jako FIFO fronty s výchozí hloubkou 2. Pro potřeby softwarové simulace jsou tyto komponenty modelovány jako fronta s nekonečnou hloubkou.

Třída `stream` je šablonová třída (podobně jako `std::vector`), a je tudíž možné ji specializovat na prakticky libovolný uživatelský datový typ (není podporována specializace na uživatelské třídy s metodami, mimo jiné i kvůli absenci virtuální tabulky metod v hardware). Pro komunikaci se stream objektem jsou dostupné čtyři metody – blokující a neblokující čtení a zápis.

Blokující čtení může způsobit pozastavení logiky, dokud je FIFO fronta na rozhraní prázdná (případně dokud není proveden handshake). Analogicky, blokující zápis způsobí pozastavení logiky, dokud je FIFO fronta plná. Pozastavení logiky znamená zastavení vykonávání celého HLS bloku, ve kterém je daný stream objekt přítomný. V případě, že je blok řetězený, zastaví se i vykonávání pipeline. Pro případy, kdy není žádoucí zablokovat vykonávání logiky, je možné použít neblokující funkce pro zápis a čtení, které pouze vracejí příznak, zda se operace podařila, na základě čehož se dá větvit navazující logika [17].

2.3 Současný stav kompilátoru

Soudobá implementace překladu $P4_{16}$ kódu do VHDL využívá architekturu navrženou již pro $P4_{14}$, pojmenovanou p4vhdl. Implementace je postavená kolem myšlenky šablon, neboť většina entit jazyka P4 má konzistentní chování nehledě na konkrétní program, a tudíž i jejich zapojení na úrovni VHDL popisu bude stále stejné, lišit se budou pouze bitové šířky a interní implementace několika málo komponent, jako je kontrolní blok a bloky akcí [2].



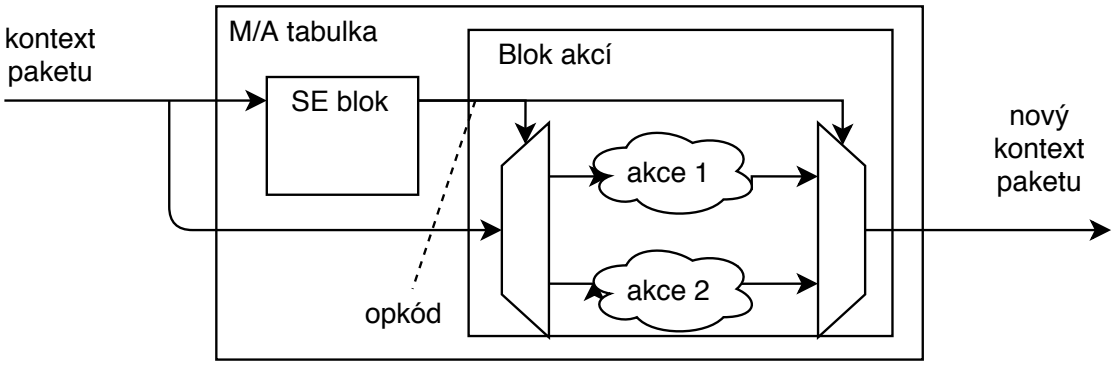
Obrázek 2.4: P4 procesní pipeline

Zjednodušeně řečeno, kontrolní bloky jsou implementovány jako posloupnost M/A tabulek a směrovačů (obrázek 2.4), přičemž tabulka je syntetizována na základě P4 kódu a směrovač realizuje podmíněné sekce v kontrolních blocích. Každá tabulka také obsahuje vlastní zjednodušený směrovací engine, který adresuje další tabulku na základě toho, zda ta aktuální nalezla pravidlo pro vstupní paket, nebo ne [2]. Jednotlivé tabulky je možné selektivně vypínat/zapínat. Vypnutí alespoň jedné tabulky zastaví celou procesní pipeline a umožňuje rekonfigurovat komponenty, jejichž chování nejde změnit atomicky.

Pakety při vstupu do zařízení procházejí komponentou zvanou parser, která rozdělí paket na dvě části – bitový vektor reprezentující všechny protokolové hlavičky definované v P4 programu a zbytek paketu (data). Pokud P4 program obsahuje vzájemně vylučující se protokoly (například TCP a UDP), pak se bitový vektor pro hlavičky naplní hodnotami přítomnými v paketu a zbytek je nedefinovaný[5]. Bitový vektor ještě navíc může obsahovat speciální metadata – položky, které nejsou parsovány přímo z paketu, ale jsou typicky využívány jako extra uložiště pro informace. Tato metadata vždy mají definovanou hodnotu.

Zatímco paketové hlavičky procházejí celou M/A posloupností (spolu s několika kontrolními signály, například zda se má paket považovat za zahozený, či zda se má na výstupu duplikovat), data paketu jsou uložena v paralelně vedených FIFO frontách. Poté, co je paket plně zpracován (tj. bitový vektor reprezentující jeho hlavičky a metadata prošel všemi M/A bloky), je paket zrekonstruován komponentou deparser, která z bitového vektoru oddělí nevalidní hlavičky, metadata a zpátky vrátí paketová data. Následně je paket odeslán na výstupní interface.

[5] Jazyk P4 má syntaktické konstrukty pro testování validity hlaviček, a to jak na úrovni podmínek v kontrolních blocích (tabulka bude aplikována pouze, pokud paket obsahuje zvolené protokolové hlavičky), tak na úrovni pravidel (konkrétní pravidlo se bude aplikovat pouze na pakety s platnými hlavičkami).



Obrázek 2.5: Zapojení bloku akcí

Akce, asociované s jednotlivými pravidly, jsou zapojeny tak, jak je naznačeno na obrázku 2.5. Kontext paketu (bitový vektor hlaviček a metadat) vstoupí do bloku s vyhledávacím algoritmem (SE blok). Pokud je nalezeno nějaké pravidlo, je nastaven `opkód` pro spuštění odpovídající akce (pokud se žádné pravidlo nedá na paket aplikovat, pak je spuštěna výchozí akce, typicky `no-op`). Akce může změnit hodnoty v kontextu paketu, přičemž tento změněný kontext je následně poslán dál v procesní pipeline.

Kompilátor p4vhdl také umožňuje užití registrů a čítačů. Tyto komponenty musí být vázané na konkrétní tabulku, nemohou existovat globálně. Z tohoto důvodu mohou být použity pouze v rámci akcí, které jsou používané tou samou tabulkou. Akce mají k těmto komponentám vyvedeny komunikační rozhraní, které, v rámci zajištění atomicity jednotlivých operací, pozastaví vykonávání celé procesní pipeline, dokud není konkrétní operace dokončena.

Tým, pracující na implementaci nového kompilátoru, byl z důvodů časové náročnosti nucen převzít většinu existujícího kódu překladače $P4_{14}$ do VHDL a upravit tyto zdrojové kódy tak, aby byly kompatibilní s novou architekturou kompilátoru a jiným programovacím jazykem (původní kompilátor vznikal v jazyce Python, nový kompilátor je napsán v C++).

Záměrem bylo zachovat původní architekturu hardwarového popisu a pouze ji postupně vylepšovat a rozšiřovat o prvky, které s ní nejsou v rozporu. Zatím jediným rozšířením bylo začlenění syntézních nástrojů pro vysokoúrovňovou syntézu (HLS) do celkového procesu překladu. Pomocí HLS je nyní implementován celý blok akcí, u kterého se očekává, že možnost popsat jejich chování pomocí vysokoúrovňového jazyka výrazně urychlí implementaci nových syntaktických konstruktů jazyka $P4_{16}$. Naopak registry a čítače nebyly převedeny do nového kompilátoru a jejich zapojení je jedním ze zaměření této práce.

MI32 rozhraní

Nestandardizované rozhraní navržené a používané sdružením CESNET. Toto rozhraní se používá pro čtení a zápis hodnot vybraných registrů v FPGA čipu ze software, typicky s cílem konfigurovat logiku obvodu za běhu aplikace. Jak plyne z názvu, rozhraní umožňuje čtení a zápis 32bitových hodnot. Adresace je definována programátorem FPGA aplikace a umožňuje adresovat jednotlivé bajty, ačkoliv v praxi se nejčastěji používá adresace po čtyřech bajtech. Signály používané při komunikaci přes MI32 sběrnici jsou popsány v tabulce 2.1 [9].

Název signálu	Šířka [bit]	Typ	Popis
DWR	32	Vstup	Data pro zápis
ADDR	32	Vstup	Adresa pro čtení/zápis
RD	1	Vstup	Požadavek na čtení
WR	1	Vstup	Požadavek na zápis
BE	4	Vstup	Maska platných bytů signálu DWR
DRD	32	Výstup	Vyčtená data
ARDY	1	Výstup	Potvrzení přijetí adresy pro čtení/zápis
DRDY	1	Výstup	Indikace platnosti vyčtených dat

Tabulka 2.1: Signály MI32 rozhraní

V rámci P4 firmwarů se MI32 používá k povolování/zakazování tabulek, ke vkládání pravidel do tabulek, případně jejich mazání. Taktéž se využívá pro čtení a zápis registrů a čítačů.

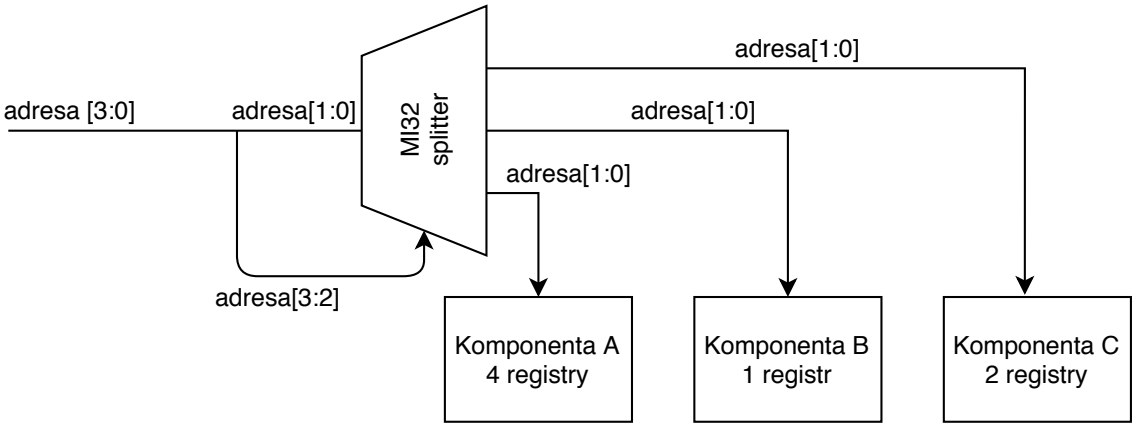
Paměťový prostor a adresace

MI32 používá jednoduchý paměťový prostor, kde každá entita přístupná na rozhraní dostane alespoň tolik adres, kolik potřebuje přímo adresovatelných registrů. Tyto adresy všechny existují v jednom spojitém bloku.

Jelikož řada adresovatelných komponent je ve VHDL hierarchicky zanořených v sobě a je nutné MI32 požadavek na základě adres směrovat do konkrétních (pod)komponent, využívají se prvky zvané MI32 splittery pro realizaci tohoto směrování. Jedná se v podstatě o jednoduché demultiplexory, které jako řídící signál mají několik málo vrchních bitů adresy a zbytek adresy spolu s daty směrují do patřičné podkomponenty.

Tento systém je velice rychlý, ale zároveň vede na neoptimální využití adresového prostoru. V praxi se totiž adresový prostor vypočítává podle následujícího algoritmu:

1. Najdi nejzanořenější komponentu ve VHDL hierarchii
   (a) Spočítej její velikost – počet adres, které potřebuje k adresaci svých registrů, zarovnej na nejbližší vyšší mocninu dvou
2. V nadřazené komponentě zjisti velikost největší podkomponenty (N)
3. V nadřazené komponentě zjisti počet adresovatelných podkomponent a zarovnej tuto hodnotu na nejbližší vyšší mocninu dvou (C)
4. Vytvoř adresový MI32 splitter, který přijímá adresu o šířce $\log_2 C + \log_2 N$. Vrchních $\log_2 C$ bitů bude použito jako select signál, zbylých $\log_2 N$ bitů bude posláno na nižší úroveň
5. Aplikuj stejný postup pro nadřazené komponenty, dokud nedorazíš ke kořeni hierarchie



Obrázek 2.6: Schéma MI32 splitteru

Obrázek 2.6 demonstruje, jakým způsobem bude adresace fungovat, dodrží-li se algoritmus uvedený výše. Pokud budeme mít tři komponenty na nejnižší úrovni hierarchie, pak nejprve spočítáme, která z nich potřebuje nejvíc adres (komponenta A — 4 adresy, tudíž 2 bity pro adresaci) a následně vytvoříme splitter, který potřebuje 2 extra bity pro zvolení správné komponenty. Celkem bude mít adresa 4 bity — dva pro adresu komponenty a dva pro adresu registru uvnitř komponenty, nehledě na to, zda daná komponenta tolik adres upotřebí, či nikoliv.

Jelikož registry a čítače v $P4_{14}$ jsou podkomponentami tabulek a tabulky všechny existují na stejné úrovni hierarchie, tak stačí, aby minimálně jediná tabulka používala příliš velké registrové pole, a tak počet adres bude velice rychle spotřebován (počet adres je teoreticky 2^{32}, nicméně MI32 pro P4 zarovnává adresy na násobky čtyř, tudíž je reálně pouze 2^{30} unikátních adres).

Device Tree

Device Tree je datová stromová struktura, navržená k popisu hardwarových periferií. Je používána v Linuxovém kernelu pro popis konfigurace počítače, díky které si kernel může v průběhu bootování zjistit, v jakém prostředí je spouštěn, a zavést patřičné ovladače. Výhoda tohoto přístupu tkví v tom, že kernel je díky tomuto přístupu kompatibilní s různými sestaveními základových desek a jejich periferií (paměti, disky, grafické čipy, atd.) [5].

Netcope využívá Device Tree pro popis adresového prostoru MI32 ve firmwarech vygenerovaných z P4 zdrojových kódů. Hlavní motivace zde není informovat kernel o schopnostech naprogramovaného FPGA čipu, cílem je především popsat dostupné entity v P4 programu a jak s nimi komunikovat (tedy jejich adresu na MI32) a libovolný uživatelský software si může při inicializaci tyto informace přečíst a patřičně se jim přizpůsobit.

Device Tree může existovat ve dvou variantách – textové a binární. Textová reprezentace je určena pro lidské uživatele, ať už pro specifikaci nebo inspekci. Binární reprezentace je určena pro praktické implementace, neboť je datově úspornější. Pro práci s binární reprezentací je určena knihovna `libfdt`, dostupná ve všech hlavních linuxových distribucí a pro převod mezi textovým a binárním formátem slouží konzolový nástroj `dtc`.

Díky tomuto systému je možné tvořit konfigurační knihovny, které jsou programově nezávislé, a není tedy nutné je přeprogramovat a překompilovat pokaždé, když uživatel přeloží nový P4 program.

Je zajímavé, že autoři specifikace jazyka v počátcích vývoje $P4_{14}$ neměli možnost, jak standardizovat informování konfiguračních knihoven o schopnostech P4 programu, a byli nuceni generovat konfigurační knihovny na míru překládaným P4 programům. Kompilátor p4c pro $P4_{16}$ již tento problém řeší skrze JSON strukturu označovanou jako P4 info, kterou umí vygenerovat ze zadaného P4 programu.

Kapitola 3

Návrh

Tato kapitola se na úvod vymezuje vůči současnému stavu architektury firmwarů, které jsou produkovány P4 kompilátorem p4vhdl. V navazujících podkapitolách jsou uvažovány různé varianty implementace externů v rámci kompilátoru p4c, které berou v potaz identifikované nedostatky. Kromě samotného návrhu zapojení externů je řešena i jejich konfigurace přes konfigurační sběrnici MI32.

3.1 Kritika současného stavu

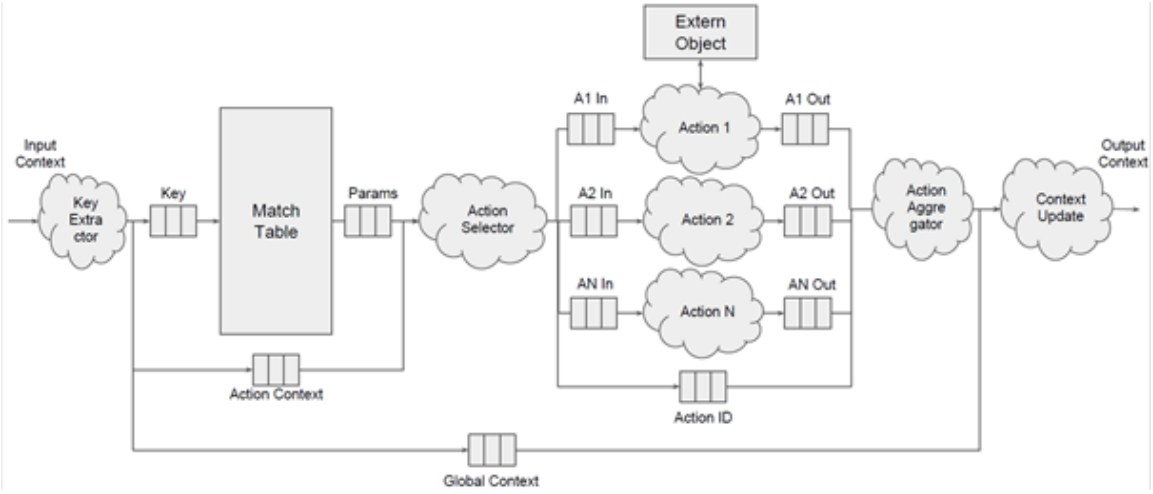
Současný stav implementace, popsaný v podkapitole 2.3, si s sebou nese hned několik základních nedostatků. Tím prvním je šířka datového kontextu, který je předán pro zpracování každé akci. Jmenovitě pro každý paket, který je zpracováván alespoň jednou tabulkou a alespoň jednou akcí, musí být do akce zaslán celý kontext paketu, tedy všechny hlavičky, které v P4 programu parsujeme, plus metadata, v případě Netcope P4[1] nazvaná `intrinsic_metadata`, obsahující jednak instrukční bity pro přepínací pole a duplikační jednotku, ale také metadata o paketu, jako je například časové razítko jeho přijetí.

Pokud tedy kupříkladu zpracováváme ethernetovou hlavičku (112 bitů), IPv4 hlavičku (160 bitů) a ke každému paketu si udržujeme jeho `intrinsic_metadata` (120 bitů), musíme do každé akce distribuovat všechna tato data, i kdyby akce reálně třeba jen dekrementovala TTL.

Jedno z možných řešení tohoto problému by vyžadovalo implementaci extrakce kontextu paketu (obrázek 3.1). Ten by nejprve z paketu vyextrahoval kontext klíče pro konkrétní tabulku a následně agregovaný kontext pro všechny akce tabulky dohromady.

Kontext klíče by byl zpracován tabulkou, která by v případě nalezení shody vytvořila kontext parametrů akce. Poté by komponenta zodpovědná za výběr a spuštění akce mohla spojit dohromady kontext parametrů a kontext dat potřebný pro aktuálně vykonávanou akci. Výsledek provedené akce by následně byl aktualizovaný kontext akce, kterým by posléze byl zaktualizovaný originální kontext celého paketu.

[1] Komerční řešení postavené na společné platformě firmy Netcope Technologies a sdružení CESNET.



Obrázek 3.1: M/A tabulka s extrakcí kontextu paketu

Tato implementace by snížila zátěž na množství vodičů ve výsledném obvodu, a to velice radikálně, když uvážíme, že v praxi jednotlivé akce pracují jen s velice omezenou množinou dat (v příkladu s dekrementací TTL bychom posílali pouze osm bitů místo tří set devadesáti dvou).

Druhým nedostatkem je práce s objekty registrů a čítačů. Implementace p4vhdl cíleně nepodporuje globální registry a čítače, ale pouze ty vázané na konkrétní tabulku. Navíc, přístup k takovému objektu zastaví vykonávání všech ostatních akcí, dokud patřičná operace čtení nebo zápisu není dokončena. Tato implementace cíleně předcházela problémům se synchronizací paralelních přístupů k takovýmto objektům. Řešení je to ovšem pomalé a zavádí nepříjemné omezení na vývojáře.

Podstatnou překážkou je i optimalizace výsledného obvodu. Kódy akcí mohou být relativně složité, s datovými závislostmi, a je poměrně složité ručně proregistrovat VHDL popis těchto akcí. Navíc pro P4 je VHDL automaticky generované a uživatel tedy ani nemá možnost editovat popis pro zajištění lepší výkonnosti. U $P4_{16}$ je kód akcí ještě mnohem složitější, především umožňuje zápis komplexnějších matematických operací, které by v podstatě nebylo možné za současného stavu dál optimalizovat.

Mezi nevýhody patří i nemožnost přečíst z čipu, jaká pravidla jsou nahraná v tabulkách. Jediný způsob, jak si efektivně pamatovat, jaká pravidla byla nahrána do tabulky (a na které řádky), je držet tyto informace v nějaké softwarové databázi. Tento problém byl již adresován několika projekty, například firma Netcope poskytuje potřebný software v rámci knihovny nazvané np4atom[2], popřípadě existuje i řešení konfigurovatelné skrze P4 Runtime protokol [13].

Za zmínku stojí i způsob práce s payloadem paketů. Jelikož je payload každého jednotlivého paketu veden v separátních FIFO frontách, není možné s ním pracovat v rámci akcí nebo třeba externů (což je potenciálně problém pro různé hashovací a šifrovací externy). Architektura současně nepodporuje paketové hlavičky proměnných délek.

[2]Součást produktu souhrnně pojmenovaného Netcope P4.

3.2 Návrh realizace nové funkcionality

Jak už bylo řečeno dříve, $P4_{16}$ zavádí řadu nových konstrukcí, se kterými se kompilátor p4vhdl nemusel potýkat. O řešení části z nich se postará již integrované využití HLS – logické a matematické výrazy, stejně jako jednoduché podmínky se dají prakticky beze změny vůči P4 programu zapsat v C++ a následně přeložit přes HLS. Navíc syntéza s využitím HLS řeší i problematiku proregistrovávání logiky, jelikož tuto operace umí dělat plně automatizovaně a často s mnohem lepšími výsledky, než jakých dosáhne sám programátor.

Návrh proto musí adresovat následující dvě témata – atomické sekce a externí objekty. Atomické sekce jsou podle názvu úseky kódu v P4, které se mají navenek provádět jako jedna operace, ideálně buď v jediném taktu hodinového signálu nebo přinejhorším tak, aby v případě delšího chování nemohla jiná akce způsobit nekonsistenci dat zpracovávaných v atomické sekci.

Externí objekty jsou naproti tomu generické komponenty jako čítače, registry nebo třeba i tabulky s velkou paměťovou kapacitou (které by se nemohly vejít na prostor FPGA čipu) a zároveň takové objekty, se kterými potenciálně mohou interagovat akce z různých tabulek.

Následující podkapitoly uvádějí několik možných variant návrhů implementace. Tyto varianty jsou seřazeny dle složitosti jejich implementace a po těchto podkapitolách následuje jejich srovnání a vybrání vhodné varianty. Všechny uvedené varianty jsou ortogonální vůči optimalizaci datové šířky signálů navržené v předchozí podkapitole.

Varianta 1 – Externy vázané na konkrétní tabulku

První varianta vychází z existující implementace pro $P4_{14}$, byť mírně upravené. Zatímco v $P4_{14}$ jsou registry a čítače komponenty sousedící s komponentou vykonávající akce a existuje mezi nimi komunikační rozhraní, zde můžeme využít možností HLS a umístit externy přímo dovnitř HLS bloku.

Každý extern by byl instanciován jako globální (statická) proměnná, ke které by měly přístup všechny akce dostupné pro danou tabulku. Toto řešení je z hlediska výkonu ideální pro aplikace, které používají extern v jedné jediné akci v jedné jediné tabulce, neboť syntézní nástroj takové chování dokáže rozpoznat a dobře optimalizovat.

Pokud by byl extern používán z více akcí v rámci jedné tabulky, řešení by stále fungovalo, nicméně syntézní nástroj by velice pravděpodobně musel plánovat nejhorší možný souběh přístupů, a proto by časování výsledného obvodu nebylo dobré.

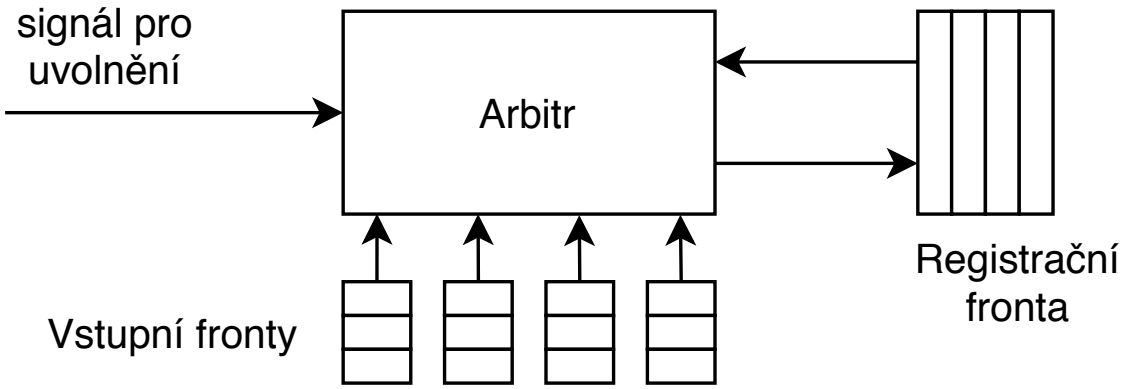
Toto řešení by navíc nedokázalo plně pokrýt standard $P4_{16}$, který umožňuje i existenci globálních extern objektů. Taktéž by v tomto zapojení nebylo možné implementovat atomické sekce, jelikož by neexistoval žádný rozumný způsob, jak zajistit výlučný přístup k objektu.

Varianta 2 – Arbitrované externy vázané na konkrétní tabulku

Předchozí variantu by bylo možné rozšířit tak, aby se dosáhlo lepšího časování a možnosti implementovat atomické sekce. Toto rozšíření by využívalo arbitry, tedy komponenty, které řídí odbavování požadavků nad sdílenými objekty.

Arbitr typicky kontroluje N přístupových bodů (například FIFO fronty) a periodicky nebo při změně vstupů vydává „one-hot“ povolení k přístupu – tedy pouze jeden vstupní bod může v daný moment komunikovat s arbitrovanou komponentou. Pokud se tedy v jeden moment objeví více současných přístupů, jeden bude odbaven a zbylé budou čekat na

udělení dalšího povolení. Běžně bude takový arbitr používat algoritmus typu round-robin pro odbavování jednotlivých front [12].



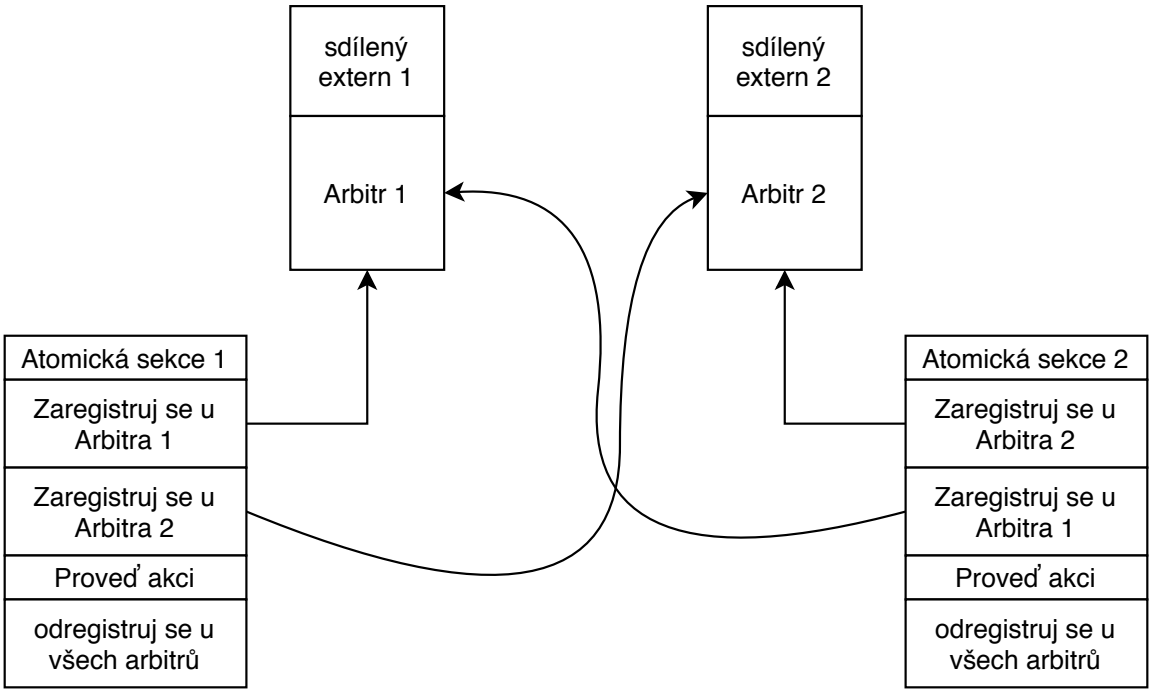
Obrázek 3.2: Arbitr externu s rozhraním pro atomické sekce

Abychom mohli implementovat atomické sekce, musíme rozšířit arbitra ještě o extra rozhraní, jaké je naznačeno na obrázku 3.2. Kromě standardních N vstupních front pro požadavky obsahuje arbitr ještě extra registrační frontu, skrze kterou je možné alternovat jeho logiku pro přidělování lístků. Požadavek v registrační frontě by obsahoval identifikátor vstupu, kterému by arbitr od okamžiku registrace výlučně generoval povolení pro přístup, a to až do chvíle, dokud by nebyl aktivován signál pro uvolnění. Aby se předešlo vyhladovění, musel by arbitr střídavě odbavovat normální požadavky a střídavě nové požadavky z registrační fronty.

Komplikuje se tím nicméně implementace logiky akce. Přístup ke sdílenému objektu přes arbitra je operace, která může potenciálně blokovat. Proto bude potřeba proces akce rozdělit do více podprocesů, které budou moci pracovat paralelně, a které budou dynamicky žádat externí objekt o data a přijatá data budou dynamicky odbavovat.

Podobným způsobem se dají řešit i atomické sekce, v rámci kterých se nepřistupuje ke sdíleným externím objektům – proces akce se rozpadne na dva podprocesy. První podproces zpracovává operace před atomickou sekcí a druhý podproces zpracovává atomickou sekci a zbytek akce, přičemž přístup do atomické sekce je blokující.

Výše uvedené řešení ovšem funguje pouze pro takové atomické sekce, které pracují pouze s jedním sdíleným extern objektem. Musíme ještě uvažovat případ, kdy atomická sekce používá více sdílených externů, aby se předešlo deadlocku.



Obrázek 3.3: Deadlock při registraci sdílených externů

V první řadě je potřeba zajistit, aby logika akce nemohla začít zpracovávat atomickou sekci, dokud není úspěšně zaregistrovaná u všech arbitrů všech objektů. Toto velice snadno vede na deadlock, jak ukazuje obrázek 3.3. V softwarové implementaci bychom tomuto problému snadno předešli modifikovaným algoritmem pro večeřící filozofy [6]. V hardwarové implementaci bych spíš navrhoval zřízení komponenty superarbitra, který by dostával požadavky na registrace množin externích objektů a velice snadno by mohl odbavovat disjunktní požadavky, zatímco kolidující požadavky by odbavoval blokující metodou FIFO.

Je však podstatné zmínit, že výše uvedené řešení může být velice drahé na zdroje na čipu. V takovém případě je možné upravit kompilátor tak, aby odmítal zdrojové P4 kódy, které by používaly v atomické sekci více jak jeden sdílený extern objekt.

Varianta 3 – Arbitrované globální externy

Poslední varianta adresuje možnost používat globální externy v rámci P4 programu, tedy objekty, které v daném kontrolním bloku může používat libovolná akce z libovolné tabulky (respektive libovolná tabulka napříč všemi kontrolními bloky). K implementaci této varianty je potřeba, aby byla implementována arbitrace z varianty 2, avšak v tomto případě už není dále možné mít externy (a jejich arbitry) instanciované uvnitř HLS bloku reprezentujícího sadu akcí konkrétní tabulky.

Externy i s arbitry by bylo nutné instanciovat na stejné úrovni, kde jsou zapojeny všechny M/A bloky a od arbitrů vyvést komunikační rozhraní do každého M/A bloku, odkud by bylo následně možné napojit toto rozhraní na HLS blok s akcemi. Přidat tuto variantu jako inkrement varianty 2 by nebylo příliš komplikované.

Díky vyšší obecnosti a extra vodičům by tato varianta poněkud zhoršila latenci komunikace s externem a bylo by vhodné ponechat i implementace předchozích variant jako optimalizace pro externy, které nejsou sdíleny mezi tabulkami.

Srovnání variant

Zatímco varianta 1 řeší poměrně úzce definovaný případ užití (externy vázané na jednu tabulku, bez možnosti využití atomických sekcí), varianta 3 řeší naopak všechny standardem dovolené případy užití. Je dobré poznamenat, že v praxi je možné implementovat všechny varianty inkrementálně, a tudíž postupně funkcionalitu rozšiřovat. Navíc, přestože varianta 3 implementuje kompletní funkcionalitu, může se ukázat být jako málo výkonná, a proto jsou předchozí varianty relevantní jako její optimalizace v jasně definovaných kontextech.

Pro potřeby této práce bylo rozhodnuto implementovat variantu 1 s cílem mít co nejdříve funkční prototyp (byť s limitovaným využitím), s možností jej v budoucnu dále rozšiřovat. Rozhodnutí bylo podpořeno i faktem, že varianta 1 je funkčně ekvivalentní k řešení z kompilátoru p4vhdl a oproti tomuto řešení umožňuje snadno rozšiřovat množinu podporovaných extern objektů. Zkušenost z praxe taktéž ukazuje, že pokročilejší případy užití nejsou mezi uživateli tak často využívané, a tak jejich implementace nemusí být tak výhodná.

3.3 Konfigurace externů přes rozhraní MI32

Řada často používaných objektů, které se ve standardu $P4_{16}$ řadí pod externy, vyžaduje konfiguraci za běhu aplikace, případně uživatel potřebuje vyčítat stav těchto objektů. Typicky se jedná o zjištění stavu čítačů nebo nastavení výchozí hodnoty registru, který se používá k výpočtu distribuce paketů do front. V komplexnějších případech může jít třeba o vložení šifrovacího klíče do kryptografického primitiva.

Na rozdíl od $P4_{14}$, kde jediné podporované externy byly registry a čítače, tedy jednoduché datové objekty, do kterých se dá přes MI32 přímo zapsat nebo naopak z nich číst, v $P4_{16}$ se od začátku počítá i s komplexnějšími externy, jejichž konfigurace, respektive čtení nemusí být tak přímočaré. Nabízí se proto výběr ze dvou variant řešení, které jsou popsané níže.

Varianta 1 – Stránkování

První varianta umožňuje zachovat stávající štědrý způsob přidělování adres, skrze jednoduché navýšení počtu dostupných adres. Toto navýšení by bylo možné díky stránkovacímu registru, který by mohl adresovat například 2^{32} stránek, z nichž každá by pojmula 2^{30} adres. Nejjednodušší implementace by pak mohla alokovat každé tabulce jednu stránku – tím pádem by většina stávající implementace hierarchie splitterů mohla zůstat nedotčená a pouze splitter na nejvyšší úrovni by místo bitů adresy používal pro select signál hodnotu stránkovacího registru.

Tato varianta by umožnila zachovat jednoduchost stávající adresace a zároveň by umožnila využití výrazně většího množství adres (dá se předpokládat, že dřív přestanou stačit zdroje na čipu, než by došlo k vyčerpání takového paměťového prostoru). Vynutila by si však komplikovanější obslužný kód v software. Ten by si buď musel pamatovat, jaká stránka je zrovna vybraná a v případě potřeby ji změnit, nebo by se hodnota stránkovacího registru musela nastavit před každou operací, což by zdvojnásobilo latenci každé jednotlivé konfigurační operace.

Varianta 2 – Dvojitá nepřímá adresace

Druhá varianta naproti tomu nevyžaduje změnu veškerého existujícího software a zároveň umožňuje ještě výrazně efektivnější využití paměťového prostoru, byť na úkor pomalejší

režie při operacích ze software. Varianta počítá s existencí čtveřice registrů na úrovni HLS bloku akcí, které by byly použity pro nepřímou adresaci jednotlivých externů a práci s nimi.

Tyto čtyři registry by byly následující:

- AOE (Address-of-extern) – registr pro adresování konkrétního externu instanciovaného v bloku akcí
- AIE (Address-in-extern) – registr pro adresaci paměti uvnitř zvoleného externu
- Data – datový registr pro zápis/čtení
- Command – řídící registr pro spouštění čtecích/zápisových operací nad externem. Zapsání hodnoty do tohoto registru by buď způsobilo čtení z vybraného externu z vybrané adresy, přičemž výsledek čtení by byl zapsán do datového registru, nebo v případě zápisu by byla hodnota datového registru poslána do externu i se zvolenou interní adresou.

Toto schéma má hned několik výhod. V první řadě je podstatně jednodušší zjistit potřebnou velikost adresového prostoru pro MI32 – ta je fixní na čtyři adresy. Za druhé, pokud pro obsluhu každého externu existují efektivně pouze dva registry – adresa uvnitř externu a data (zbylé registry slouží pro režii), pak se na úrovni C++ šablon v HLS dá jednoduše definovat společný interface, a výrazně se tudíž zjednodušuje emitování HLS kódu. Přidělení adres externům je taktéž relativně jednoduché, jelikož si je může libovolně vygenerovat kód emitující HLS a pouze je nutné vygenerovanou hodnotu propagovat do Device Tree.

Na druhé straně tu jsou i nevýhody. Nepřímá adresace výrazně zvyšuje režii – místo jediné operace čtení/zápisu je nyní nutné provést čtyři operace – zapsat adresu externu, zapsat adresu uvnitř externu (ta může být teoreticky nepovinná), zapsat data a zapsat do příkazového registru (respektive nejdřív zapsat příkaz a pak vyčíst data). Je nutné podotknout, že pokud by se zachovala přímá adresace, ale otázka adresového prostoru byla řešena stránkováním, byla by režie stále problém, byť potenciálně menší.

Kontrolní software pro zápis pravidel do tabulek by se v tomto případě nemusel měnit, komunikace s nimi by nebyla ovlivněna. Pouze by bylo nutné rozšířit kontrolní software o podporu nových externů – vyčtení jejich adres, velikosti a typu z Device Tree a implementace konfiguračního algoritmu, která by byla jasně daná tvůrcem konkrétního externu.

Srovnání variant

Z obou variant byla vybrána varianta 2 – dvojitá nepřímá adresace. Důvodem je především menší dopad na stávající kontrolní software, který v této variantě není třeba přepsat, ale pouze rozšířit o nové typy externů. Výhodou je i efektivnější práce s paměťovým prostorem MI32. Nevýhodou je náročnější režie. V praktickém nasazení se ovšem klade mnohem větší důraz na výkon při vkládání pravidel, než na vyčítání převážně statistických údajů z registrů a čítačů.

3.4 Shrnutí

V rámci návrhu byly diskutovány různé varianty implementace napojení externů na akce popsané v HLS a taktéž varianty komunikace s těmito externy přes MI32 sběrnici. Vybrána byla varianta nearbitrovaných externů vázaných na konkrétní tabulku, které budou adresované přes MI32 skrze dvojitou nepřímou adresaci. Tyto dvě varianty umožňují rychle vytvořit prototyp, které bude možné dále iterativně vylepšovat a rozšiřovat.

Kapitola 4

Implementace

Tato kapitola přibližuje praktickou implementaci podpory externích objektů do kompilátoru `p4c-p4vhdl`, respektive podporu překladu $P4_{16}$ programů do VHDL skrze kompilátor `p4c`. Implementace postupuje podle variant návrhu zvolených v předchozí kapitole.

Členění do podkapitol je následující. Nejdříve jsou popsána specifika architektury nového kompilátoru a práce s grafovou reprezentací překládaných programů, a jakým způsobem je možné z této reprezentace získat data potřebná pro úspěšnou instanciaci externích objektů. Zbylé podkapitoly se věnují popisu implementace základních externích objektů v HLS, jejich integrace do projektu kompilátoru a také napojení na konfigurační sběrnici MI32.

4.1 Práce s kompilátorem

Jak již bylo řečeno v předchozích kapitolách, kompilátor p4c pracuje se strukturou IR – Intermediate Representation, což je orientovaný acyklický graf. Programový back-end kompilátoru je navržen tak, aby s využitím návrhového vzoru Návštěvník šlo implementovat algoritmus, který se aplikuje pouze na jasně specifikovaný typ uzlu. Možná je i práce nad množinu uzlů, pokud je algoritmus implementován nad obecným uzlem, ze kterého jsou poděděné další podtypy.

Kód 4.1 předvádí textovou reprezentaci IR uzlu obsahujícího deklaraci registru. Registr je jedním z nejjednodušších externích objektů jazyka $P4_{16}$ – jedná se o pole dané délky s buňkami se specifikovanou bitovou šířkou a metodami `read`/`write` pro přístup k jednotlivým buňkám pole (specifikace chování těchto metod je popsána v hlavičkovém souboru v1model.p4 [1]). Objekt z deklarace byl v původním programu deklarován jako `register<bit<16>>(1) reg`. Tato deklarace říká, že objekt je pojmenován `reg`, má jednu jedinou buňku pole a ta buňka je číslo o šířce 16 bitů (P4 standard dovoluje registru ukládat i jiná data, než jenom čísla, specifikace však postrádá jiný datový typ, který by se dal ukládat).

Když se vrátíme zpět k ukázce IR v kódu 4.1, můžeme si všimnout, že všechny potřebné informace jsou zde obsaženy. První řádek odpovídá uzlu typu `Declaraction_Instance` a jeho atribut name obsahuje hodnotu `reg`, což je název instanciovaného objektu (řetězec `reg_0/` je pomocný a není přímo obsažen v atributu).

Následně si můžeme povšimnout uzlu s typem `Type_Specialized`, s poduzlem `baseType` typu `Type_Name` v rámci nějž je uložena informace, že název typu je `register`. Argumenty typu (tedy specializace šablony) je pole obsahující jedinou položku – číslo o šířce 16 bitů.

```
[88160] Declaration_Instance name=reg_0/reg declid=188
  annotations: [88162] Annotations
    [88164] Annotation name=name needsParsing=0
      [88168] StringLiteral value=ingress.reg
        type: [876] Type_String
  type: [1231] Type_Specialized
    baseType: [1227] Type_Name
      path: [1226] Path name=register absolute=0
    arguments: [1230] Vector<Type>
      [269] Type_Bits size=16 isSigned=0
  arguments: [82665] Vector<Argument>
    [9773] Argument name=<null>
      expression: [9770] Constant value=1 base=10
        type: [1] Type_Bits size=32 isSigned=0
```

Kód 4.1: IR podstrom deklarace registru

O řádek níže můžeme najít argumenty konstruktoru, což je opět pole o jedné položce, obsahující konstantu 1.

ExternDeclarationInspector

Se znalostí struktury uzlu pro deklaraci externího objektu je možné naimplementovat jednoduchý inspektor, který z deklarace dokáže důležité hodnoty extrahovat, jak znázorňuje kód 4.2. Můžeme si povšimnout, že první dvě `preorder` metody vrací `true`, protože vektor uzlů obsahující parametry šablony je podgrafem uzlu `Type_Specialized` a ten je zase podgrafem uzlu `Declaration`. Pokud by tyto dvě metody vrátily `false`, zastavilo by se jejich provedením další prohledávání a nikdy by se neprovedly metody pro získání argumentů šablony a konstruktoru.

Zatímco název cílového typu externu je převeden na pouhý řetězec, argumenty šablony a konstruktoru jsou ponechány v podobě pole uzlů. To proto, že tyto uzly nemusí být jenom konstanty, ale třeba i výrazy, a proto je ponecháno na uživateli třídy, aby data interpretoval tak, jak potřebuje.

Během implementace již existovaly třídy inspektorů, které bylo možné aplikovat na výrazy a získat jejich finální HLS reprezentaci. Inspektor znázorněný v kódu 4.2 byl taktéž rozšířen o metodu `getHls`, která vrací kompletní HLS reprezentaci instanciace registru. Pro IR graf uvedený v kódu 4.1 by taková instanciace v HLS vypadala jako `register<ap_uint<16>, 1> reg;`. Můžeme si všimnout, že P4 datový typ `bit` se změnil na `ap_uint` (tedy funkčně evivalentní datový typ v HLS). Také si můžeme všimnout, že velikost registrového pole (hodnota 1) je na rozdíl od P4 kódu parametrem šablony a nikoliv konstruktoru. Důvodem je fakt, že tato hodnota musí být statická, jelikož v HLS nejde použít dynamická alokace paměti.

ExternMap

Aby bylo možné inspektor deklarací použít, bylo nutné vytvořit třídu pojmenovanou jako ExternMap. Tato třída je inspektor, který je na program aplikován jako poslední krok v midend fázi překladu. Účelem třídy je projít všechny kontrolní bloky programu, nalézt

```
class ExternDeclarationInspector : public Inspector {
public:
    std::string m_externName = "";
    std::string m_externType = "";
    const IR::Vector<IR::Type> *m_templateArgs = nullptr;
    const IR::Vector<IR::Argument> *m_ctorArgs = nullptr;

    bool preorder(const IR::Declaration *node) override {
        m_externName = /* Convert node name to string */;
        return true;
    }

    bool preorder(const IR::Type_Specialized *node) override {
        m_externType = /* Convert node->baseType->path name to string*/;
        return true;
    }

    bool preorder(const IR::Vector<IR::Type> *value) {
        m_templateArgs = value;
        return false;
    }

    bool preorder(const IR::Vector<IR::Argument> *value) {
        m_ctorArgs = value;
        return false;
    }
};
```

Kód 4.2: Inspektor pro extrakci informací o externím objektu z IR uzlu deklarace

všechna použití extern objektů v akcích a akumulovat následující informace o jednotlivých externech:

- Podgraf IR s deklarací externu
- Jména akcí, v nichž je extern použity a jména tabulek, které dané akce používají
- Identifikátory atomických sekcí, v nichž je extern použit

Na základě těchto informací je možné kontrolovat, zda je extern sdílený mezi tabulkami (je použit v jedné akci, která je ovšem používaná dvěma tabulkami), v kolika akcích je použit a v kterých atomických sekcích je použit. Tyto kontroly umožňují zamezit překladu programů, které by nebyly validní vůči zvolené implementační variantě 1 (3.2).

Zapojení do kompilátoru

Jak již bylo řečeno, midend aplikuje instanci inspektoru ExternMap na celý kód a následně tento inicializovaný inspektor předá backendu pro další zpracování. Backend při emitování cílových kódů pro M/A tabulky zkontroluje, zda daná tabulka nepoužívá nějak problematický extern objekt (například takový, který by byl sdílený mezi více tabulkami) a následně při emitování HLS kódů pro blok akcí může využít ExternDeclarationInspector pro korektní instanciaci externu v HLS kódu. Taktéž byl dopsán dedikovaný inspektor pro korektní emitování HLS kódu pro volání metod nad extern objektem.

Jelikož bylo prioritou mít co nejdříve funkční prototyp, byl backend omezen tak, aby bylo možné úspěšně přeložit pouze takové P4 programy, v nichž je každý extern použit právě v jedné akci v právě jedné tabulce. Extern tím pádem není jakkoli sdílen, což výrazně urychlilo vývoj první verze prototypu.

4.2 Zapojení konfiguračního rozhraní MI32 do bloku akcí

Při integraci konfigurace přes MI32 bylo nutné vyřešit především to, jak správně zapojit komunikační rozhraní. Top metoda HLS bloku obsahovala před integrací pouze dva parametry – vstupní a výstupní stream rozhraní pro pakety. Logika top metody při každém svém spuštění blokujícím čtením vyčetla paket ze vstupní fronty, provedla na něm transformace odpovídající P4 programu, a nakonec ho blokujícím zápisem vložila do výstupní fronty.

Jelikož je nutné odbavovat MI32 požadavky nezávisle na paketech, byly do rozhraní top metody přidány další dvě stream rozhraní pro požadavky a odpovědi na MI32. A protože komunikace na MI32 pracuje s větším množstvím signálů se specifickým chováním, byla vytvořena také pomocná komponenta nazvaná obálka MI32. Tato komponenta mapuje MI32 signály na rozhraní HLS streamů. Pro každý příchozí požadavek vyčte adresu, typ požadavku (čtení/zápis) a data (nedefinovaná v případě zápisu). Obálka zařídí správnou signalizaci přijetí požadavku a vyčtená data vloží do vstupního streamu HLS bloku. Pokud byla požadovaná operace čtení, HLS blok vloží do výstupního streamu čtyři bajty dat, tato jsou vyčtena řadičem a správným způsobem odvysílána po MI32.

Logika HLS bloku byla upravena tak, aby na svém začátku zkusila neblokujícím čtením vyčíst MI32 požadavek ze vstupního streamu. Pokud se čtení podaří, je tento požadavek prioritně odbaven a pak následuje běžná logika pro zpracování paketů. V rámci té se musí pro čtení příchozích paketů taktéž použít neblokující operace. Zápisy do výstupních streamů byly nicméně ponechány jako blokující. V případě MI32 požadavků není možné ze

software posílat dotazy rychleji, než jsou v hardware odbavovány, a tudíž výstupní stream je implementován jako jednoduchý registr. V případě paketů se výstupní fronta, která je na rozhraní streamu připojena, může teoreticky naplnit. Praktická implementace ovšem musí zajistit, že pakety je vždy možné zapsat na výstup (tj. žádné se neztratí) a zároveň, že tento zápis nezablokuje vykonávání zřetězené pipeline HLS bloku.

Z tohoto důvodu byla zavedena jednoduchá heuristika pracující nad celou komponentou HLS bloku. Sleduje se počet paketů ve vstupních a výstupních frontách HLS bloku a za předpokladu, že by vložení dalšího paketu do vstupní fronty mohlo znamenat, že se po zpracování nepodaří zapsat do výstupní fronty, heuristická komponenta nastaví signály tak, aby se vstupní fronta HLS bloku jevila jako plná, a tudíž nebylo možné do ní cokoliv vložit. Tento přístup zajistí, že HLS blok může vždy fungovat, nikdy neblokuje na přístupech do streamů, a proto je MI32 komunikace vždy úspěšně odbavovaná.

Implementace má jeden nedostatek, zatím pouze teoreticky diskutovaný. Přestože je odbavování MI32 požadavků nezávislé na odbavování paketů (pomineme-li atomický přístup k externům), a mohlo by tudíž fungovat plně paralelně, HLS momentálně nemá standardizované syntaktické konstrukty, jak toto dát překladovému nástroji najevo[1]. Z tohoto důvodu i pokud logika obsluhující MI32 bude mít menší latenci, než obsluha paketů, bude syntézním nástrojem zarovnána na stejnou hodnotu, aby se dalo snadno vyřešit časování pipeliningu. Kvůli tomu se ale může stát, že pokud bude P4 akce obsahovat opravdu komplexní logiku (jejíž VHDL ekvivalent bude mít velikou latenci), může obsluha MI32 požadavku zabrat tolik času, že vyprší časový limit pro odbavení a odpovědi na čtecí požadavky[2] se nikdy nedoručí, přestože logika jako taková funguje. Vypršení limitu by navíc bylo pravděpodobně vyhodnoceno jako chyba a celé zařízení by pak bylo restartováno.

Uvedený nedostatek je momentálně pouze akademickou teorií a zatím ani neexistovaly prostředky, jak ho docílit v praxi na uživatelské aplikaci. Dá se předpokládat, že čím více read/modify/write operací nad registrovým externem bude konkrétní P4 akce obsahovat, tím spíš problém nastane, jelikož tyto operace zaberou minimálně 1 takt[3]. Taktéž se tato situace předpokládá pouze na FPGA čipech od firmy Intel, které časový limit uměle zavádějí; na FPGA čipech od firmy Xilinx není známo, že by limitace existovala[4].

4.3 Implementace externu Register

Kód 4.3 ukazuje pseudokód implementace chování externu typu Register dle specifikace v1model[5].

Konstruktor a metody `read` a `write` odpovídají specifikovanému chování a jsou dostupné pro volání z P4 programu. Všimněme si, že třída je parametrizovaná parametrem `<type>`, neboť registrové buňky mohou být teoreticky libovolného typu. Použitá verze v1model má pouze jeden použitelný datový typ a tím je `bit<size>`, tedy celočíselná proměnná se

[1] V době psaní textu tuto podporu má pouze nástroj Quartus HLS, implementace diplomové práce nicméně probíhala s nástrojem Vivado HLS.

[2] Zápisové požadavky by proběhly v pořádku, protože řadič by včas přijal data, odeslal potřebnou signalizaci, a HLS blok by pak měl tudíž dostatek času na odbavení.

[3] Délka taktu se odvíjí od cílené propustnosti firmware. Například pro propustnost 80Gbps se používá frekvence hodin 160MHz.

[4] Limitace není nutně vlastností čipu, spíše může být dána implementací ovladačů pro komunikaci přes PCI, respektive napojení MI32 na PCI, které je na kartách od firmy Intel vedena přes rozhraní Avalon a CCIP.

[5] Během psaní práce se změnila verze v1model a s ní i rozhraní registrového externu. Tato diplomová práce počítá s původním návrhem, tj. číslo verze menší než 20200408.

```
Class Register<type> {
    Array<type> data;

    Function read(out <type> result, in UINT32 index) {
        If (index < data.length) result = data[index];
    }

    Function write(in UINT32 index, in <type> value) {
        If (index < data.length) data[index] = value;
    }

    Register(in UINT32 size) {
        data.resize(size);
    }
}
```

Kód 4.3: Pseudokód implementace read/write metod P4 Registru dle specifikace v1model

specifikovanou bitovou šířkou. Chování metod `read` a `write` je nedefinované, pokud zadaný index nespadá do mezí pole.

HLS kód implementující výše uvedené chování musí navíc ještě obsahovat dvě metody s následujícími signaturami:

```
void mi32_write (uint32_t address, uint32_t data);
uint32_t mi32_read (uint32_t address);
```

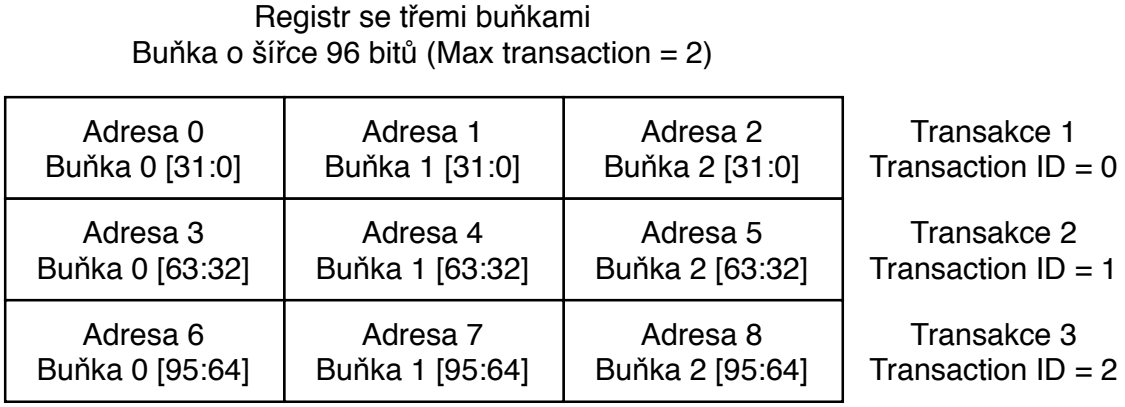
Zatímco metody `read` a `write` budou volané pouze z HLS reprezentace P4 akcí, výše uvedené metody budou volané v rámci obsluhy MI32 požadavků a měly by být součástí rozhraní každé třídy, která implementuje P4 extern v HLS. Chování těchto dvou metod znázorňuje kód 4.4.

Pseudokód `mi32_*` metod je relativně komplikovaný v porovnání s těmi standardními. Je to proto, že šířka buňky v registrovém poli může být libovolná, a tudíž může být i širší, než kolik se vejde do jedné čtyřbajtové transakce. V takových případech je potřeba několika čtyřbajtových transakcí na realizaci libovolného jednoho čtení nebo zápisu.

Kód si tudíž musí poradit s otázkou, jak vyřešit atomicitu a adresaci jednotlivých transakcí. Atomicita je v implementaci vyřešena jednoduchým cache registrem, přičemž jak zápis, tak čtení komunikuje pouze s tímto registrem a až ve chvíli, kdy jsou transakce hotové, je tento registr atomicky propsán do konkrétní buňky (respektive tato buňka je propsána do registru v případě čtení). Cache registr musí mít z principu stejnou bitovou šířku jako buňky registrového pole.

```
function mi32_write(in UINT32 address, UINT32 data) {
    if (TRANSACTION_ID > MAX_TRANSACTION_ID) return;
    elif (address < data.length) cache = data;
    else {
        cache |= data << TRANSACTION_ID*32;
    }

    if (TRANSACTION_ID == MAX_TRANSACTION_ID) {
        data[address] = cache;
    }
}

function mi32_read(in UINT32 address) -> UINT32 {
    if (address < data.length) {
        cache = data[address];
        Return cache[31:0];
    }

    if (TRANSACTION_ID > MAX_TRANSACTION_ID) return 0;

    return cache[31 + TRANSACTION_ID*32 : TRANSACTION_ID*32];
}
```

Kód 4.4: Pseudokód pro zpracování MI32 transakcí



Obrázek 4.1: Schéma adresace registrů s buňkami širšími, než 32 bitů

Otázka adresace ovšem nemá tak přímočarou odpověď. Jsou možné dva přístupy – buď budou mít všechny transakce stejnou cílovou adresu, a tudíž metody musejí obsahovat extra logiku, která detekuje kompletní sekvenci transakcí a taktéž správně propisuje hodnotu cache do datových buněk a naopak, nebo je možné každé transakci alokovat unikátní adresu, ze které půjde spočítat začátek a konec sekvence transakcí. Možná adresace je v takovém případě následující – každá buňka registrového pole se virtuálně rozdělí na několik podbuněk, velkých právě 32 bitů (poslední podbuňka může být i menší).

Druhé navrhované řešení je úspornější na plochu na čipu i na složitost logiky. Je navíc flexibilnější v případech, kdy by softwarová aplikace musela být restartována uprostřed sekvence transakcí. Jelikož neexistují registry, které by sledovaly průběh transakcí a vše lze vypočítat, je velice jednoduché dostat se zpět do konzistentního stavu. Z těchto důvodů byl pro implementaci zvolen právě tento přístup.

Způsob adresace názorně vysvětluje obrázek 4.1 na příkladu s tříbuňkovým registrem, kde každá buňka má 96 bitů. Když každou buňku rozdělíme na čtyřbajtové podbuňky, dostaneme tři podbuňky pro každou buňku pole. Adresa podbuňky P registrové buňky B v registrovém poli o velikosti N se spočítá následovně: $ADDR = P * N + B$. První transakce v sekvenci je tudíž taková, kdy $P = 0$ a poslední je taková, že $P = \lceil W/32 \rceil - 1$ (W je šířka každé jednotlivé buňky v bitech). Tento způsob adresace zachovává klasické indexování $0....N - 1$ pro registry s buňkami o šířce 32 bitů nebo méně.

Bylo by možné paměťový prostor i otočit tak, aby související podbuňky byly adresovány sousedními adresami (na obrázku 4.1 by tudíž buňka 0 měla adresy 0, 1, 2 místo 0, 3, 6), tato varianta ale nebyla shledána jako dostatečně intuitivní.

4.4 DeviceTree

Aby bylo možné použít MI32 komunikaci pro konfiguraci registrového externu z uživatelské aplikace, je nutné znát přesnou adresu a parametry registru, se kterým má aplikace komunikovat. Toto je stejně jako u všech ostatních P4 entit zajištěno přidáním odpovídajících informací do Device Tree, které popisuje vlastnosti firmwaru naprogramovaného skrze P4.

Vzhledem ke stavu implementace se všechny externy nacházejí v paměťovém prostoru uvnitř tabulky, uvnitř HLS bloku, který realizuje logiku všech akcí používaných tabulkou. Všechny externy jsou nepřímo adresovatelné přes tu samou čtveřici registrů. Kód 4.5 ukazuje, jakým způsobem byl rozšířen záznam v Device Tree, který reflektuje tyto skutečnosti a zároveň nechává prostor pro budoucí rozšiřování.

```
externs {
    offset=<0x80>
    size=<0x4>

    registers {
        r_0 {
            reg-name = "reg";
            address = <0x0>;
            size = <0x1>;
            width = <0x10>;
        };

        r_1 {
            reg-name = "reg2";
            address = <0x1>;
            size = <0xa>;
            width = <0x30>;
        };
    };
};
```

Kód 4.5: Sekce Device Tree pro extern objekty, obsahující dva Registry

Všechny externy jsou obsaženy v sekci s názvem `externs`, a společným parametrem pro všechny externy je taktéž adresa (offset), na které se nachází první ze čtyř společných registrů (počet registrů je obsažen v atributu `size`, ale jeho hodnota bude vždy stejná).

Následně si povšimněme podsekce `registers`, která obsahuje všechny instance konkrétního externu typu Register. Atribut `reg-name` identifikuje jméno použité v P4 programu, adresa značí hodnotu, přes kterou se objekt nepřímo adresuje, `size` označuje počet buněk a `width` označuje šířku buněk (v bitech). Přidávání dalších externů by v budoucnu znamenalo přidat další podsekci s informacemi relevantními pro konfiguraci konkrétního typu externu.

4.5 Shrnutí

V rámci této kapitoly byla popsána implementace podpory jednoduchého P4 externu typu Register (dle specifikace v1model) do $P4_{16}$ kompilátorového backend `p4c-p4vhdl`. Cílem bylo co nejdříve se dopracovat k prototypu, který bude otevřen budoucím rozšířením, ale zároveň bude poskytovat alespoň minimální užitečnou funkcionalitu. Výsledkem je implementace, která umožňuje konkrétní instanci externu použít v jediné akci v jediné tabulce, a která je navíc konfigurovatelná ze software skrze MI32 sběrnici. Uživatelské softwarové aplikace mohou přítomnost tohoto externu ve firmware detekovat a adresovat díky záznamům v Device Tree.

Kapitola 5

Experimenty a testování

Tato kapitola prezentuje sadu experimentálních scénářů, které byly použity pro zjištění výkonnostních parametrů implementace a jsou diskutovány i optimalizační kroky, realizované v reakci na prvotní výsledky měření. Dále kapitola popisuje testovací metodologii, použitou pro ověření funkčnosti implementovaného řešení.

5.1 Experimenty

Cílem experimentů bylo ověřit praktické fungování implementovaného řešení a změřit některé jeho základní parametry. Mezi tyto parametry patří – latence, inicializační interval, využití zdrojů, časování. Parametry byly převzaty z odhadů nástroje Vivado HLS, které jsou generované jako jeden z výstupů syntézy HLS kódu. Cíl pro časování syntézy byl v projektu nastaven na 4ns.

Praktické fungování bylo odzkoušeno vůči síťové kartě NFB-200G2QL[1], která dosahuje propustnosti až 200Gbit/s na dvou QSFP28 rozhraních na běžném síťovém provozu a pracuje na frekvenci 200MHz. Karta je osazená FPGA čipem Virtex7 UltraScale+ od firmy Xilinx a se softwarovou aplikací komunikuje skrze PCI Express 3. generace se dvěma šetnáctilinkovými endpointy (teoretická rychlost komunikace je až 2x128Gbps). V době psaní textu ještě integrace $P4_{16}$ nebyla na dostatečně úrovni, aby dovolila praktické měření propustnosti.

Následuje popis experimentálních scénářů (každý scénář je pojmenován podle odpovídajícího P4 programu v příloze na CD):

- **src0-noreg** – referenční scénář, který neobsahuje žádný registr, pouze výchozí akci tabulky (`p_update`), která přepíše hodnotu `ethertype` v hlavičce paketu na 0. Tento scénář poskytuje referenční bod pro měřené parametry obvodu.
- **src1-onereg-rw** – úprava předchozího scénáře. V programu je instanciován jeden registr s jedinou buňkou o šířce 16 bitů. Do paketu do hlavičky `ethertype` se tentokrát zapisuje aktuální hodnota registrové buňky, která se následně inkrementuje o 1. Oproti referenčnímu scénáři je očekáván pokles výkonu o přibližně 25 %. Hodnota je dána zkušeností s chováním firmwarů přeložených z kompilátoru pro $P4_{14}$.
- **src2-onereg-r** – variace na předchozí scénář, bez inkrementace registrové buňky (provádí se tedy pouze čtení z registru a zápis do hlavičky paketu).

[1] https://www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx

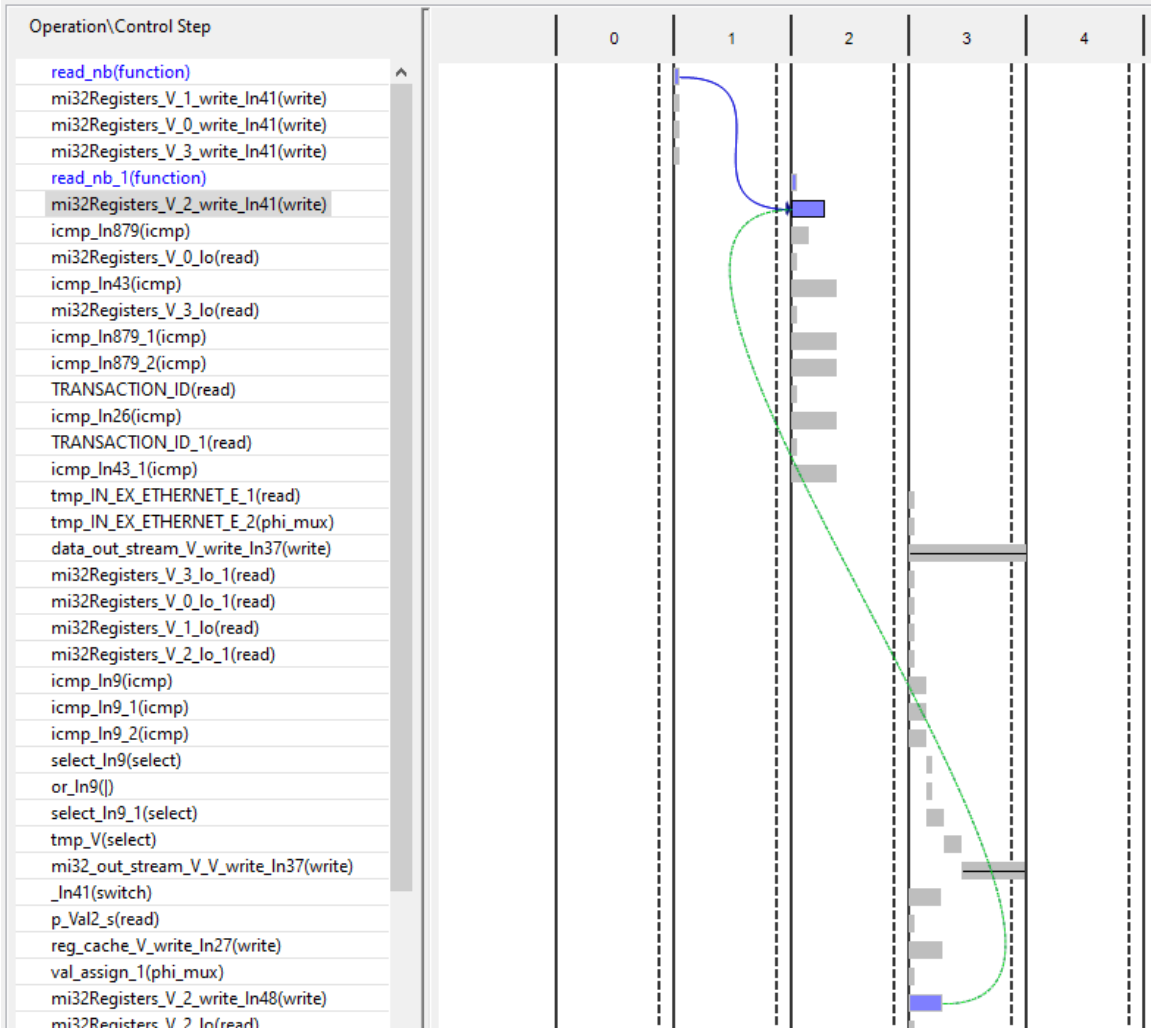
- **src3-onereg-w** – variace na scénář src1, neobsahuje čtení a zápis do paketové hlavičky, pouze do registrové buňky zapíše aktuální hodnotu `ethertype` paketu.
- **src4-tworegs-two-actions** – přidává další registr s jednou buňkou o šířce 48 bitů a další akci (`p_update2`), která pracuje stejně, jako `p_update`, nicméně místo hlavičky `ethertype` manipuluje s hlavičkou `source MAC`. Tyto dvě akce nikdy nemohou být spuštěny zaráz, a proto pravděpodobně nebudou mít vliv na výkon obvodu, nicméně obslužný MI32 kód pro registr s buňkami širšími, než 32 bitů bude komplikovanější. Dopad na výkon se dá odhadovat na přibližně 50 %. Opět je toto očekávání založeno na zkušenosti s chováním $P4_{14}$ firmwarů.
- **src5-tworegs-two-tables** – rozpad předchozího scénáře tak, že každý registr je používán v jiné tabulce. Jednodušší (užší) registr se používá v tabulce `filter`, zatímco ten širší se používá v tabulce `filter2`. Scénář může lépe ukázat dopady různě širokých registrů na výsledný výkon.
- **src6-tworegs-sequential** – v rámci jedné akce jsou použity dva registry se stejně širokými buňkami tak, aby jejich použití nutně vedlo na sekvenční logiku – do prvního registru se zapíše hodnota `ethertype`, následně se tato hodnota přepíše čtením z druhého registru a ten se následně inkrementuje. Cílem je zjistit vliv takového programu na latenci a inicializační interval výsledného obvodu.
- **src7-tworegs-parallel** – variace na předchozí scénář, tentokrát je možné logiku paralelizovat.
- **src8-onereg-big-narrow** – tento program používá velké registrové pole (1000 položek) a funguje velice podobně jako src1, akorát s tím rozdílem, že registrovou buňku indexuje přes hodnotu `ethertype`. Registrové buňky mají šířku 16 bitů. Od tohoto a následujícího scénáře se očekává přesunutí registrového pole do BRAM pamětí a s tím spojený pokles výkonu.
- **src9-onereg-big-wide** – úprava předchozího scénáře tak, aby registrové buňky měly šířku 48 bitů.
- **src10-onereg-two-actions, src11-onereg-two-tables** – tyto scénáře jsou nepřeložitelné a byly použity pouze pro ověření správného chování kompilátoru. Scénář 5 obsahuje jednu tabulku se dvěma akcemi, které obě přistupují k jednomu registru. Druhý scénář má dvě tabulky sdílející jednu akci, která používá ten samý registr. Jelikož jsou oba programy nepřeložitelné, nebudou do srovnávací tabulky zahrnuty.
- **src12-onereg-super-wide** – tento scénář byl přidán později v návaznosti na optimalizační kroky diskutované níže. Využívá registr se šířkou buňky 96 bitů, což vyžaduje tři MI32 transakce pro nakonfigurování/přečtení.

Úvodní měření nad testovacími scénáři odhalilo, že všechny varianty, které obsahovaly jakýkoli registr, měly inicializační interval 2, zatímco varianta bez externu měla inicializační interval 1. Tabulka 5.1 ukazuje konkrétní srovnání prvních dvou scénářů.

Označení	Časování (ns)	Pipelining (cykly hodin)		Využití zdrojů (počty)			
		Latence	Interval	FF	LUT	BRAM	DSP
src0-noreg	2.136	4	1	425	493	0	0
src1-onereg-rw	2.946	4	2	432	286	0	0

Tabulka 5.1: Srovnání časování a využití zdrojů pro první dva scénáře

Zvýšení inicializačního intervalu nebylo očekávané a při bližším zkoumání vyšlo najevo, že k němu došlo kvůli zpětné vazbě na jednom z kontrolních registrů pro MI32. Obrázek 5.1 ukazuje snímek Ganttova diagramu s vyznačenou zpětnou vazbou, zatímco pseudokód 5.1 demonstruje konstrukci v kódu, která ke vzniku zpětné vazby vedla.



Obrázek 5.1: Ganttův diagram se zpětnou vazbou

```
if (mi32_stream_in.read_nb(request)) {
    mi32registers[request.address] = request.data;
    if (request.address == COMMAND_REG) {
        /* Logic */
```

```
        mi32registers[DATA_REG] = data;
    }
}
```

Kód 5.1: Pseudokód demonstrující zpětnou vazbu

Zpětná vazba vyjadřuje datovou závislost typu WAW, kdy se zápis do datového registru musí projevit dřív, než se v dalším taktu pokusí pipeline zapsat data následujícího požadavku do registru, který potenciálně může být taktéž datový. V reálné aplikaci by se tak nicméně dít nemělo – jednak nedává smysl do datového registru zapisovat ve dvou následujících taktech, a zároveň požadavky určitě nebudou přicházet tak často.

Jelikož ale stejně není potřeba, aby šel příkazový registr číst, bylo možné kód 5.1 přepsat do podoby kódu 5.2, která eliminuje zpětnou vazbu. Tím se docílilo, že do paměti se zapisuje pouze jednou během konkrétního průchodu pipeline a ačkoliv se to může dít v různých větvích logiky, samotný zápis jde naplánovat do stejného okamžiku. Toto řešení efektivně odstraňuje příkazový registr, jelikož požadavky přicházející po MI32 už ani nejde zapsat nikam jinam. Zároveň ale fyzická přítomnost takového registru nebyla nikdy zcela nutná – všechny operace musejí probíhat v konstantním čase, a tudíž nevystává potřeba požadavek nejprve zadat a následně se dotazovat tohoto konkrétního registru na úspěšné dokončení operace.

```
if (mi32_stream_in.read_nb(request)) {
    if (request.address == COMMAND_REG) {
        /* LOGIC */
        mi32registers[DATA_REG] = data;
    }
    else {
        mi32register[request.address] = request.data;
    }
}
```

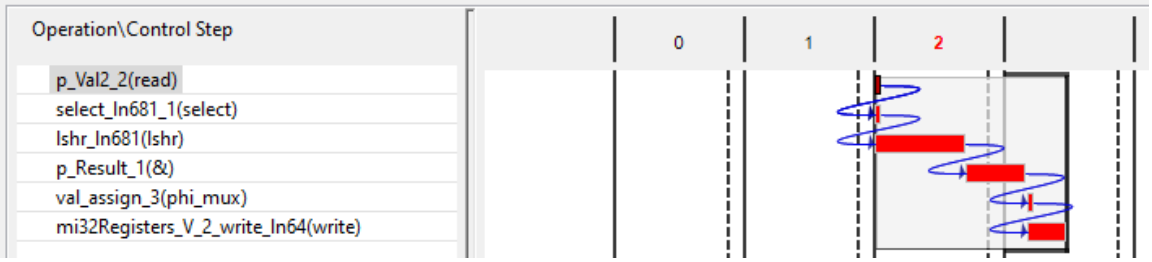
Kód 5.2: Pseudokód s eliminovanou zpětnou vazbou

Po provedení výše uvedené optimalizace a opětovném proměření byla odhalena další neefektivita implementace. Ta se projevila ve čtvrtém scénáři, který obsahuje registr širší, než 32 bitů. Tabulka 5.2 ukazuje, jak se změnily naměřené hodnoty před a po optimalizaci.

Označení	**Časování (ns)**	**Pipelining (cykly hodin)**		**Využití zdrojů (počty)**			
		Latence	**Interval**	**FF**	**LUT**	**BRAM**	**DSP**
src4-tworegs-two-actions (před optimalizací)	4.370	4	2	790	1481	0	0
src4-tworegs-two-actions (po optimalizaci)	7.208	4	1	866	1457	0	0

Tabulka 5.2: Srovnání parametrů čtvrtého scénáře před a po eliminaci zpětné vazby

Již časování původní varianty je výrazně horší, než u varianty s registrem menším, než 32 bitů. V tomto případě byl ovšem cíl časování porušen o více, než tři nanosekundy, což je mnoho. Na obrázku 5.2 je snímek ganttova diagramu, který znázorňuje příslušnou kritickou cestu.



Obrázek 5.2: Ganttův diagram s kritickou cestou

Ganttův diagram vede na konkrétní část výpočtu v metodě `mi32_read` v rozhraní třídy `Register`. Konkrétně se jedná o návratový výraz, který extrahuje 32 bitů z cache pro registrové buňky na základě zaslané adresy. Kód 5.3 ukazuje tento návratový výraz. Výpočet v kódu se snaží o dvě věci. V první řadě se snaží spočítat index vrchního a spodního bitu pro rozsah, který má být extrahován. V druhé řadě je nutné zkontrolovat, zdali vrchní limit rozsahu nepřesahuje reálnou šířku registru, ze kterého čteme, a pokud ano, pak je nutné hodnotu upravit, aby nepřetékala.

```
unsigned transHi = 32 + TRANSACTION_ID * 32;
return uint32_t(
    cache.range(
        (cache.length() > transHi ? transHi : cache.length()
    ) - 1,
    TRANSACTION_ID * 32)
);
```

Kód 5.3: Výpočet pro extrakci rosahu bitů pro MI32 transakci

Komplikaci to způsobuje ze dvou důvodů. Tím zásadnějším je nutnost provést bitový posun pro převedení extrahovaného rozsahu na `uint32_t`. V druhé řadě onen ternární operátor pro kontrolu vrchního limitu dále obfuskuje logiku. Kontrolu vrchního limitu je ovšem potřeba udělat pouze v takovém případě, kdy se provádí poslední čtecí transakce (tj. $TRANSACTION_ID == MAX_TRANSACTION_ID$). Kód 4.4 z kapitoly 4 šel díky tomu upravit způsobem, jež je vyjádřen pseudokódem 5.4.

```
TRANSACTION_ID = address / SIZE;
BITS_LOW = TRANSACTION_ID * 32;

if (address < SIZE) { // prvni transakce
    cache = cells[address];
    return cache[31 : 0];
}
else if (TRANSACTION_ID < MAX_TRANSACTION_ID) {
    BITS_HIGH = 31 + BITS_LOW;
    return cache[BITS_HIGH : BITS_LOW];
}
else if (TRANSACTION_ID == MAX_TRANSACTION_ID) {
    return cache[cache.bitsize() - 1 : BITS_LOW];
}
```

```
return 0;
```

Kód 5.4: Pseudokód s pozměněným výpočtem pro extrakci bitů

Výsledné řešení zlepšilo časování na 3.670 ns a také využití zdrojů na 797 flip-flop registrů a 897 lookup tabulek. Především počet lookup tabulek se snížil téměř na polovinu. Uvedená optimalizace bude podávat dobré výsledky pro všechny registry, u nichž bitová šířka jedné buňky nepřesáhne 64 bitů. Pro větší registry bude výsledné časování opět horší, neboť pro prostřední transakce bude nutné aplikovat bitový posun. Z tohoto důvodu byl přidán experimentální scénář 12. Tento scénář potvrdil obavy z horšího časování, které je prakticky stejné, jako ve scénáři 4 před optimalizací.

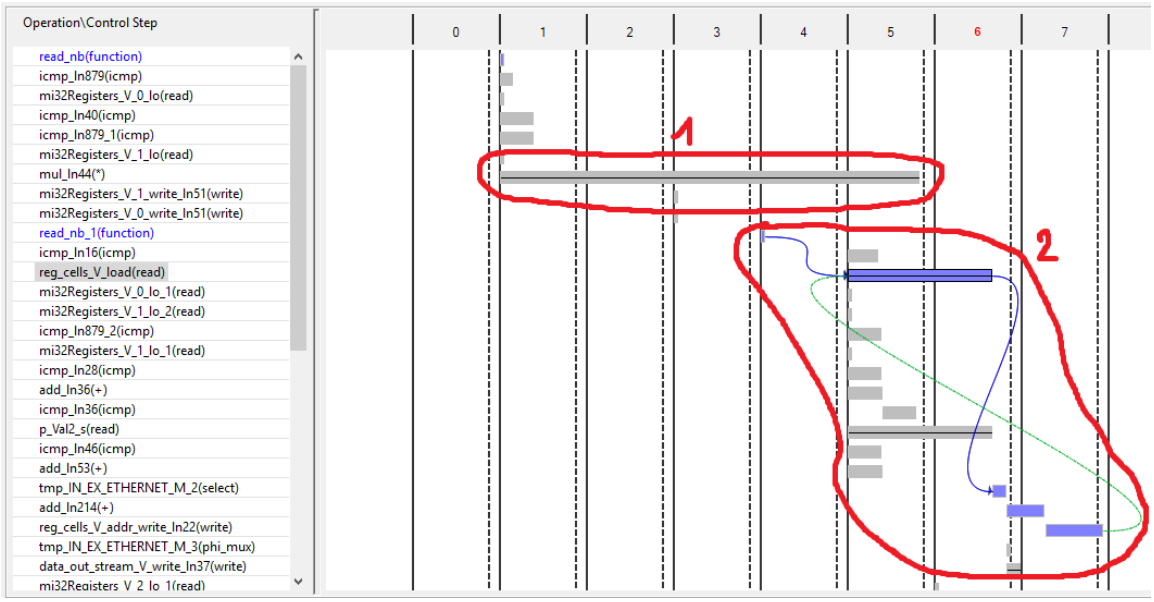
Jediné řešení této situace by bylo vytvořit specializované třídy pro různé šířky registrových buněk, respektive různé třídy pro různé počty transakcí, které jsou nutné ke kompletnímu přečtení registrové buňky. Každá z těchto tříd by pak mohla implementovat jednoduchou kaskádu podmínek, z nichž každá větev by extrahovala konstantně definovaný rozsah bitů, což by syntézní nástroj mohl převést na jednoduchý demultiplexor. Současná implementace je po optimalizacích dostatečně výkonná i pro registry s buňkami širokými až 64 bitů, což pokrývá drtivou většinu známých uživatelských aplikací.

Zajímavé výsledky poskytly i scénáře 8 a 9. Tyto obsahují velká registrová pole (konkrétně o tisíci buňkách). Dle očekávání byly využity BRAM pro implementaci paměti. Využití zdrojů ovšem ukázalo i alokaci DSP bloků pro výpočty. Výrazně se zhoršilo časování, latence, i inicializační interval (tabulka 5.3).

Označení	**Časování (ns)**	**Pipelining (cykly hodin)**		**Využití zdrojů (počty)**			
		Latence	**Interval**	**FF**	**LUT**	**BRAM**	**DSP**
src8-onereg-big-narrow	4.800	7	2	1309	802	1	4
src9-onereg-big-wide	7.783	7	2	1609	1221	3	4

Tabulka 5.3: Naměřené hodnoty pro scénáře s velkými registrovými poli

Důvodem pro zhoršení časování byl právě přístup do pomalejší paměti, který odhalil zpětnou vazbu mezi zápisem do registru v akci `p_update` a čtením tohoto registru pomocí MI32 sběrnice, případě další akcí v pipeline. Důvodem pro horší časování a využití DSP bloku byl výpočet id aktuální transakce při přístupu přes MI32. To se dělo skrze dělení, které je z pochopitelných důvodů pomalé. Obě tyto skutečnosti jsou pozorovatelné na Ganttově diagramu na obrázku 5.3. Oblast označená číslem 1 ukazuje pomalé dělení, oblast označená číslem 2 odpovídá zpětné vazbě na přístupech do registrových buněk.



Obrázek 5.3: Ganttův diagram s vyznačeným výpočtem dělení (1) a zpětnou vazbou na registrové buňce (2)

Dělení jde optimalizovat poměrně snadno – pokud uživatel specifikuje jako velikost registrového pole nějakou mocninu dvou, lze dělení nahradit jednoduchým vymaskováním a ořezáním čísla. Toto bylo ověřeno s využitím registrového pole o 1024 prvcích. Jeho použití výrazně vylepšilo latenci v obou případech, eliminovalo DSP bloky a taktéž ušetřilo některé další zdroje. Bohužel se nepodařilo eliminovat zpětnou vazbu na registrech, která ovlivňuje i inicializační interval obvodu.

5.2 Testování

Testování implementace proběhlo na několika úrovních. V první řadě vznikly unit testy napsané ve frameworku Google Test, které jsou součástí překladače a jsou rutinně spouštěny v rámci standardního vývoje. Příloha na CD obsahuje modelové P4 programy, jimiž jsou inspirované testové scénáře. Modelové programy byly též použity pro experimenty nad naimplementovaným řešením (více níže).

V druhé řadě byla provedena simulace a co-simulace nad syntetizovaným RTL obvodem v nástroji Vivado HLS. Pro tvorbu simulačního scénáře byl použit P4 zdrojový kód `src1-onereg-rw.p4`, respektive HLS kód vygenerovaný P4 překladačem. Testování nejprve přes MI32 zapíše hodnotu do jediné dostupné registrové buňky, následně pošle paket skrz akční blok, kde je zpracován skrze akci `p_update`. Očekávané chování počítá s tím, že do konkrétní hlavičky paketu bude zapsána aktuální hodnota registrové buňky a následně přečtení hodnoty registrové buňky přes MI32 vrátí inkrementovanou hodnotu.

Dále vzniknul verifikační scénář implementovaný v proprietálním frameworku sdružení CESNET a firmy Netcope, který testuje použití registru v kompletním zapojení včetně všech komponent nadřazených nad akcí. Tento verifikační scénář parsuje pouze ethernetovou hlavičku a přepisuje pole ethertype na aktuální hodnotu uloženou v registrové buňce, načež inkrementuje tuto hodnotu. Verifikace pošle několik paketů skrze pipeline a porov-

Označení	Časování (ns)	Pipelining (cykly hodin)		Využití zdrojů (počty)			
		Latence	Interval	FF	LUT	BRAM	DSP
src0-noreg	2.136	4	1	434	286	0	0
src1-onereg-rw	2.946	4	2	425	493	0	0
src2-onereg-r	2.946	4	2	425	450	0	0
src3-onereg-w	2.946	4	2	409	446	0	0
src4-tworegs-two-actions	4.370	4	2	790	1481	0	0
src5-tworegs-two-tables (filter)	2.946	4	2	425	493	0	0
src5-tworegs-two-tables (filter2)	4.314	4	2	611	1328	0	0
src6-tworegs-sequential	4.439	5	2	627	1245	0	0
src7-tworegs-parallel	4.370	4	2	677	1392	0	0
src8-onereg-big-narrow	4.800	8	2	1502	877	1	4
src9-onereg-big-wide	7.783	8	2	1760	1230	3	4
src12-super-wide	4.373	4	2	865	2119	0	0

Tabulka 5.4: Odhady časování a využití zdrojů před optimalizacemi

nává hodnoty paketů na výstupu s referenčními hodnotami. Tyto verifikace jsou typicky spouštěné pokaždé, když se do projektu integruje nová feature.

Verifikování MI32 komunikace nebylo možné zahrnout do uvedeného frameworku – ten je převzat z verifikací $P4_{14}$ aplikací a jednak v něm není implementován interface konfigurace externů jako takových, a navíc daný interface nepodporuje registry (podporuje pouze čítače a metry). Proto byla využita existující šablona napsaná v System Verilog, která byla rozšířena o provedení MI32 zápisů a čtení, přičemž výsledky čtení jsou kontrolovány vůči očekávaným výstupům. Tato verifikace není nikterak automatizovaná a je možné ji spustit pouze ručně.

Jediná možnost, jak testovat celé zapojení komplexně, je framework pro integrační a akceptační testy firmy Netcope, který pracuje s hotovými firmwary. Tento framework má tu výhodu, že nevyžaduje využití vysokoúrovňových interfaců, ale je možné v něm vytvořit i test pracující pouze s nízkoúrovňovými nástroji pro přímou komunikaci s MI32. V tomto frameworku byla proto vytvořena sada testů, která může být zahrnuta do automatického testování, jakmile bude produkt řádně integrován.

5.3 Shrnutí

V rámci testování a experimentování nad implementovaným řešením bylo odhaleno několik možností optimalizace. Jejich vliv na výkonnost výsledného řešení demonstrují tabulky 5.4 a 5.5, z nichž první obsahuje odhady využití zdrojů a časování před optimalizacemi, zatímco druhá po aplikaci všech diskutovaných optimalizací.

Optimalizace ve většině případů dokázaly vylepšit inicializační interval, latenci, a dokonce i ušetřit některé zdroje. Ukázalo se ovšem, že větší registrová pole mají velice negativní efekt na výsledné časování a nezanedbatelný vliv na výkon mají i neparalelizovatelné akce.

Označení	Časování (ns)	Pipelining (cykly hodin)		Využití zdrojů (počty)			
		Latence	Interval	FF	LUT	BRAM	DSP
src0-noreg	2.136	4	1	402	251	0	0
src1-onereg-rw	3.670	4	1	568	482	0	0
src2-onereg-r	3.670	4	1	552	453	0	0
src3-onereg-w	3.670	4	1	536	442	0	0
src4-tworegs-two-actions	3.670	4	1	797	897	0	0
src5-tworegs-two-tables (filter)	3.670	4	1	568	482	0	0
src5-tworegs-two-tables (filter2)	3.547	4	1	665	820	0	0
src6-tworegs-sequential	4.439	4	2	548	1273	0	0
src7-tworegs-parallel	3.670	4	1	700	827	0	0
src8-onereg-big-narrow	4.800	4	2	398	517	1	4
src9-onereg-big-wide	7.783	4	2	570	1386	3	4
src12-super-wide	7.355	4	1	932	2133	0	0

Tabulka 5.5: Odhady časování a využití zdrojů po optimalizacích

Měření taktéž ukázala, že implementace arbitrů je prakticky nezbytná pro jakékoli praktické použití. V případech, kdy bylo časování porušeno o více, než jednu nanosekundu, by se výsledný blok nedal automatizovaně funkčně zapojit do P4 pipeline při cílové frekvenci 200 Gbps. Byly již pozorovány případy z praxe na $P4_{14}$ programech, kdy nesplněné časování o pár desetin nanosekundy (při 100°C) vedlo na firmware, který se velice snadno zasekával, případně byla jeho propustnost snížena o více, než třetinu.

Kapitola 6

Závěr

Migrace kompilátoru jazyka $P4_{14}$ na novější verzi $P4_{16}$ představuje řadu nových výzev k vyřešení. V tomto textu byly popsány především výzvy týkající se kompilace uživatelských akcí – transformačních operací nad vybranými příchozími pakety. Tyto výzvy se týkají nejen nových typů instrukčních bloků v akcích, jako jsou například atomické sekce, ale také komunikace s externími objekty s neznámou implementací.

Byla navržena možná řešení rozebraných problémů, přičemž tato řešení byla uvažována s ohledem na syntézní nástroj HLS, který umožňuje překládat do VHDL konstrukce napsané ve vyšších programovacích jazycích jako C a C++. Navrhovaných řešení byla implementovaná velice limitovaná podmnožina, která se snaží pokrýt funkcionalitu dostupnou v rámci kompilátoru $P4_{16}$. Implementované řešení je nicméně otevřené budoucímu rozšíření na plnou podporu funkcionality $P4_{16}$.

Práce rovněž prezentuje optimalizační techniky při práci s nástrojem Vivado HLS a také způsob práce s kompilátorem p4c pro překlad $P4_{16}$ programů. Diskutované optimalizace byly proměřitelné pouze na úrovni syntézního nástroje, funkční firmware v době psaní práce nebylo možné sestavit.

Literatura

[1] Barefoot Networks, I. *V1model.p4*. 2016. [Online], [cit. 2020-04-17]. Dostupné z: https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4.

[2] BENÁČEK, P. *Generation of High-Speed Network Device from High-Level Description*. Praha, 2016. Diplomová práce. Fakulta informačních technologií na ČVUT.

[3] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N. et al. *P4: Programming Protocol-Independent Packet Processors. ACM SIGCOMM Computer Communication Review*. 2014, sv. 3, č. 44, s. 87–95. ISSN 0146-4833.

[4] DAMAJ, I. *High-level Synthesis. Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, inc. 2008, sv. 3, s. 1495–1504.

[5] devicetree.org. *DeviceTree Specification Release 0.3*. 2020. [Online], 13. Únor 2020. [cit. 2020-03-03]. Dostupné z: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf.

[6] DIJKSTRA, E. W. *Hierarchical ordering of sequential processes*. In: Hoare, C. A. R. a PERROT, R., ed. *Operating Systems Techniques*. Academic, 1972, s. 72–93.

[7] FPGA, I. *Intel HLS Compiler System of Tasks*. 2019. [Online], 8. Květen 2020. [cit. 2020-05-22]. Dostupné z: https://www.youtube.com/watch?v=aH1O4DZnjf8.

[8] GAMMA, E., HELM, R., JOHNSON, R. a Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0201633612.

[9] MATOUŠEK, J. *Implementace a verifikace vstupních a výstupních toků*. 2009. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[10] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L. et al. *OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review*. 2008, sv. 2, č. 38, s. 69–74.

[11] MEYER, B. *Object-Oriented Software Construction; 2nd ed.* Prentice Hall, 1997. ISBN 9780136291558.

[12] MICHAEL, S. S. *Simple Priority Arbiter*. All About Circuits, 2020. [Online], 11. Únor 2019. [cit. 2020-04-16]. Dostupné z: https://www.allaboutcircuits.com/technical-articles/simple-priority-arbiter-allocating-resources-embedded-systems-vhdl-logism/.

[13] NERUDA, J. *Vzdálená konfigurace P4 zařízení*. 2018. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[14] The P4 Language Consortium. *$P4_{16}$ Portable Switch Architecture*. The P4 Language Consortium, 2018. [Online], Verze 1.1.0 (2018), [cit. 2019-12-23]. Dostupné z: https://p4.org/p4-spec/docs/PSA-v1.1.0.html.

[15] The P4 Language Consortium. *P4 Language Specification*. The P4 Language Consortium, 2018. [Online], Verze 1.1.0 (2018), [cit. 2019-04-29]. Dostupné z: https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html.

[16] The P4 Language Consortium. *P4 Language Specification*. The P4 Language Consortium, 2018. [Online], Verze 1.0.5 (2018), [cit. 2019-04-29]. Dostupné z: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[17] Xilinx. *Vivado Design Suite UserGuide, High-level Synthesis*. Xilinx, 2019. [Online], [cit. 2020-05-19]. Dostupné z: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.