VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Petr Panák

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY
DEPARTMENT OF MICROELECTRONICS

VYSOKOÚROVŇOVÁ SYNTÉZA ČÍSLICOVÝCH OBVODŮ V OBLASTI SÍŤOVÝCH APLIKACÍ POPSANÝCH V JAZYCE P4

HIGH LEVEL SYNTHESIS IN NETWORK APPLICATIONS DESCRIBED USING P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Petr Panák**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Lukáš Fujcik, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Mikroelektronika a technologie**

Ústav mikroelektroniky

Student: Petr Panák ***ID:*** 191473
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Vysokoúrovňová syntéza číslicových obvodů v oblasti síťových aplikací popsaných v jazyce P4

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s principy vysokoúrovňové syntézy číslicových obvodů a dostupnými nástroji, zejména Vivado HLS a Intel HLS.
Seznamte se s jazykem P4 a jeho využitím v oblasti síťových aplikací. Zaměřte se na akce a externí bloky jazyka P4 a navrhněte vhodný způsob jejich popisu na úrovni jazyka C/C++.
Proveďte vysokoúrovňovou syntézu navržených bloků a porovnejte výstupy dosažené různými HLS nástroji (Vivado, Intel). V závěru práce zhodnoťte dosažené výsledky a diskutujte další pokračování práce.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

doc. Ing. Jiří Háze, Ph.D.
předseda rady studijního programu

ABSTRAKT

Vysokoúrovňová syntéza se stala přívětivou metodou pro návrh digitálních obvodů. Její výhodou, oproti návrhu na behaviorální úrovni, je vyšší míra abstrakce a rychlejší verifikace. To zaručuje rychlejší návrh, jenž snižuje náklady na vývoj. Tato bakalářská práce se zabývá návrhem akcí, externích bloků a přístupu rozhraní MI32. Jednotlivé komponenty návrhu jsou popsány pomocí programovacího jazyka C/C++ a syntetizován kompilátorem Intel HLS.

KLÍČOVÁ SLOVA

Vysokoúrovňová syntéza, FPGA, P4, Akce, Externí bloky, MI32, Intel HLS

ABSTRACT

High-level synthesis is a compelling method of designing a digital circuit. High abstraction and faster verification are advantages which aren't pressent in Register Transfer Level designing. That guarantees faster designing with lower development costs. This bachelor thesis deals with a digital design of actions, extern blocks and MI32 interface access. Each component design is described using C/C++ programming language and synthesised with Intel HLS compiler.

KEYWORDS

High-level Synthesis, FPGA, P4, Actions, Extern blocks, MI32, Intel HLS

PANÁK, Petr. *Vysokoúrovňová syntéza číslicových obvodů v oblasti síťových aplikací popsaných v jazyce P4*. Brno, 2020, 57 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Vysokoúrovňová syntéza číslicových obvodů v oblasti síťových aplikací popsaných v jazyce P4“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu semestrální práce panu Doc. Ing. Lukáši Fujcikovi, Ph.D. za trpělivost, odborné vedení, konzultace a podnětné návrhy k práci. Dále bych chtěl poděkovat odbornému konzultantovi Ing. Tomáši Martínkovi, Ph.D. za konzultace, příkladné vedení a velmi obohacující rady nejen k práci. Poděkování si také zaslouží má rodina a blízké okolí za jejich podporu.

Brno ....

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Chytrý řadič síťového rozhraní (anglicky Smart Network Interface Controller, zkratkou SmartNIC) je jednou z mnoha domén, kde programovatelná hradlová pole (anglicky Field Programmable Gate Array, zkratkou FPGA) hrají značnou roli i přes svou vysokou cenu. Přízni se těší díky své výkonnosti, možnost přeprogramování i po uvedení do provozu poté zaručuje flexibilitu. Programování zařízení však není jednoduché a návrh obvodů na behaviorální úrovni (anglicky Register-Transfer Level, zkratkou RTL) je často zdlouhavý i pro zkušeného programátora. Časově efektivnější způsob návrhu představuje vysokoúrovňová syntéza (anglicky High Level Synthesis, zkratkou HLS).

HLS používá pro popis funkcionality převážně programovací jazyky jako C, C++ a SystemC, které nabízí větší úroveň abstrakce. Samotná abstrakce zmenšuje pravděpodobnost chyb návrháře a urychluje návrh obvodů. Další výhodou HLS je rychlejší verifikace a možný průzkum mikro-architektur, kterých se při návrhu může vytvořit více. Na návrháři poté zůstává, zdali se rozhodne pro co nejvyšší rychlost či co nejmenší využití zdrojů. Nástroje pro HLS jsou jak komerční (Catapult C), tak neplacené od výrobců FPGA (Intel HLS, Vivado HLS).

Již zmíněné pragramovací jazyky (C, C++ a SystemC) nedosahují dostatečné abstrakce v oblasti popisu síťových prvků. Dostatečné abstrakce dosahuje programovací jazyk P4 (Programming Protocol-Independent Packet Processors), který je platformě-nezávislý a je převeden na platformě-závislý program, jenž je specifikován na danou výpočetní jednotku. Příkladem platformě-závislého programu je kompilátor P4-to-VHDL, vyvíjený v rámci sdružení CESNET, jenž převádí popis síťového prvku z jazyka P4 do jazyka VHDL (VHSIC Hardware Description Language).

Vytvoření kompilátoru však vyžaduje velké úsilí a zabere spoustu času. Propojení s nástroji HLS tedy představuje efektivní možnost jak popsat některé komponenty, například akce a externí bloky.

Cílem práce je právě popis akcí, externích bloků a přístupu rozhraní MI32 na úrovni jazyka C/C++. Tento popis je syntetizován pomocí kompilátoru Intel HLS, avšak už ne pomocí Vivado HLS.

V teoretické části práce (1) jsou popsány základní principy a optimalizační schopnosti HLS. V další části je popsán jazyk P4 s jeho základními konstrukcemi. Následně je popsán kompilátor P4-to-VHDL, jeho základní komponenty, rozhraní MI32 a přínos propojení jazyka P4 a HLS. V závěru teoretické části jsou uvedena specifika kompilátoru Intel HLS. V praktické části práce (2) je nejprve popsán koncept návrhu akcí, externích bloků a přístupu rozhraní MI32. Následně jsou popsány jednotlivé komponenty a jejich reálná implementace v HLS. V rámci vyhodnocení výsledků 3 je popsáno testovací prostředí a jsou zhodnoceny výsledky navržených komponent.

1 Teoretický úvod

V rámci teoretického úvodu je nejprve popsána problematika vysokoúrovňové syntézy. Dále je popsán jazyk P4 a některé jeho konstrukce, kompilátor P4-to-VHDL s rozhraním MI32 a jsou uvedeny možnosti, které přináší propojení jazyka P4 a vysokoúrovňové syntézy. Na závěr jsou uvedena specifika kompilátoru Intel HLS.

1.1 Vysokoúrovňová syntéza

Vysokoúrovňová syntéza (dále jen HLS) je automatizovaný proces návrhu, kdy je z algoritmického popisu chování vytvořen digitální hardware. K popisu se běžně používají programovací jazyky C, C++, SystemC a následně je v procesu syntézy provedeno několik kroků, jenž jsou popsány v rámci této kapitoly, která vychází především z [1, 3].

1.1.1 Principy vysokoúrovňové syntézy

Počínaje vysokoúrovňovým popisem aplikace, přes RTL knihovnu komponent a specifické omezující podmínky (anglicky constraints), provádí HLS nástroj následující kroky:

1. Kompilace specifikací programu.
2. Tvorba formálního modelu – diagram kontrolních a datových toků.
3. Plánování operací na hodinové cykly.
4. Přiřazování operací k funkčním jednotkám, proměnných k paměťovým elementům a hran k sběrnicím.
5. Alokace zdrojů – funkční jednotky, úložné komponenty, sběrnice a jiné
6. Generování RTL architektury.

Procesy alokace, plánování a přiřazení jsou na sobě závislé. Pakliže prioritou výsledného návrhu bude jeho rychlost, plánování operací bude ovlivňovat alokaci zdrojů, tedy nástroj upřednostní rychlost před spotřebou zdrojů návrhu. Naopak tomu bude u prostorově omezeného plánování, kde alokace zdrojů ovlivňuje plánování operací, tedy nástroj upřednostní spotřebu zdrojů před rychlostí návrhu. Při přiřazování se poté mohou vytvářet dodatečné zdroje, tedy přiřazení zdrojů ovlivňuje jejich alokaci.

1.1.2 Kompilace a tvorba formálního modelu

V první fázi se provede optimalizace kódu (např. eliminace nedosažitelného kódu, vyhodnocení výrazů s konstantami) a vytvoří se syntaktický strom na základě jednotlivých příkazů. Ze syntaktického stromu se následně vytvoří vhodná reprezentace

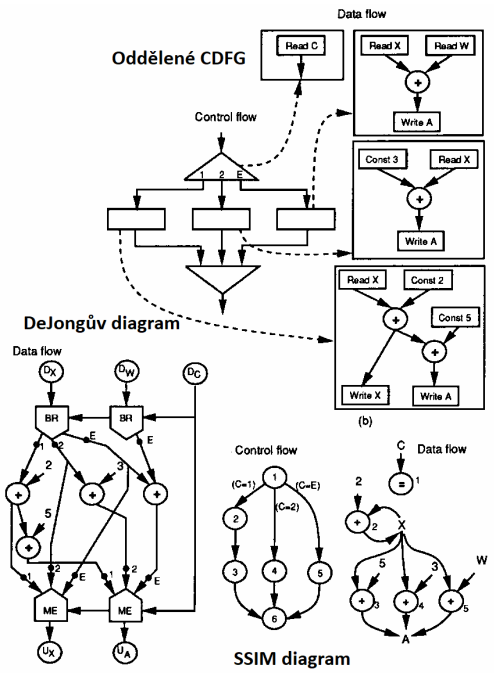
obvodu (formální model). Vhodnou reprezentací se v případě HLS staly diagramy řídicích a datových toků.

Diagramy řídicích a datových toků

Diagram datových toků (anglicky Data Flow Graph, zkratkou DFG) snadno znázorňuje závislost dat, kde každý uzel představuje operaci a hrany mezi uzly představují datové závislosti. Výška diagramu poté značí počet kroků nutných k vykonání výpočtu. DFG však neobsahuje podmíněné vyhodnocování, jenž je běžnou součástí většiny návrhů. Použití DFG jako reprezentace, by se tedy omezilo pouze na pár aplikací. Je tedy nutné vyhodnocovat i řídicí konstrukce, k jejichž reprezentaci slouží diagram řídicích toků.

Diagram řídicích toků (anglicky Control Flow Graph, zkratkou CFG) znázorňuje všechny možnosti, jak by mohl být proces vykonán v závislosti na řídicích podmínkách (podmíněné příkazy a cykly). Každý uzel v diagramu reprezentuje základní blok (tj. přímočarý kus kódu bez skoků nebo skokových cílů), hranu značící start bloku a hranu ukončující blok.

Diagram řídicích a datových toků (anglicky Control Data Flow Graph, zkratkou CDFG) spojuje funkce DFG a CFG, tedy popisuje jak řídicí konstrukce, tak datové závislosti, čímž se stává vhodnou reprezentací obvodu [2]. Obrázek 1.1 zobrazuje několik možných druhů CDFG.



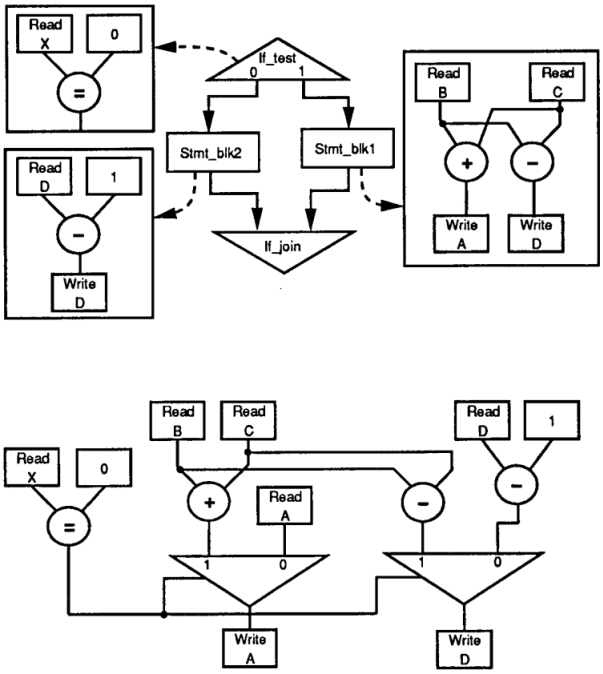
Obr. 1.1: Druhy diagramu řídicích a datových toků (převzato z [3]).

Oddělené CDFG znázorňuje DFG a CFG odděleně. DeJongův diagram ukládá kontrolní informace dovnitř DFG. U SSIM diagramu je CFG kopií DFG a ukazuje provádění jednotlivých operací.

Základní transformace

Nežli se přejde k dalším operacím, dojde k možné redukci výšky diagramu, což může vést k většímu paralelismu a zrychlení výpočtu. K tomu se využívá případných komutativních, distributivních a asociativních vlastností operátorů, které lze takto přeuspořádat.

Dalším krokem je transformace řídicího toku do datového (anglicky flattening). To zapříčiní, že všechny sekce jsou vykonávány paralelně a výsledek je vybrán multiplexorem (viz obrázek 1.2). Náročnost na zdroje je poté vyšší, jelikož nelze zdroje mezi jednotlivými větvemi sdílet.



Obr. 1.2: Transformace řídicího toku do datového (převzato z [3]).

Těmito kroky je vytvořena finální formální reprezentace v podobě CDFG, nad kterou se provádí další operace.

1.1.3 Plánování operací

Plánování je úloha, která rozděluje vstupní CDFG do podgrafů, které jsou vykonávány v jednom kontrolním kroku (taktu hodinového signálu). Operace v podgrafu

určují počet a typ funkčních jednotek, které musí být k dispozici v jednom kontrolním kroku. Funkční jednotky lze sdílet mezi více kontrolními kroky.

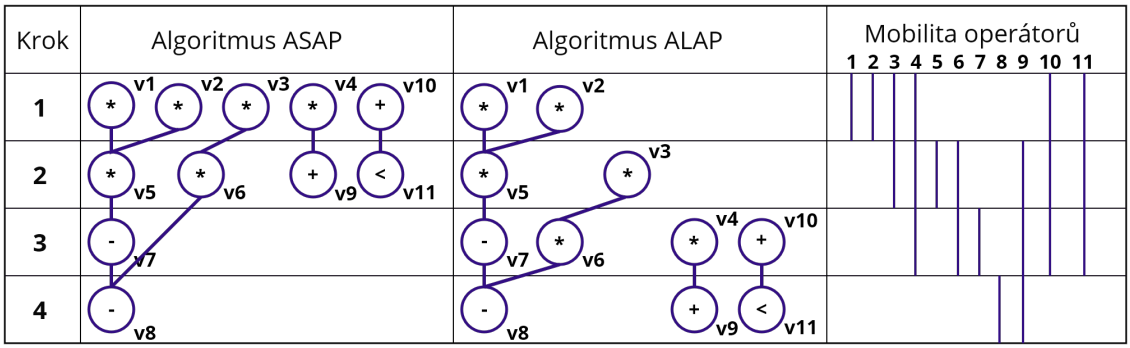
Základní plánovací algoritmy

Základními plánovacími algoritmy jsou ASAP (As Soon As Possible) a ALAP (As Late As Possible). Jejich značnou nevýhodou je neomezenost dostupných funkčních jednotek. I přesto jsou vhodné jako pomocné algoritmy při plánování složitějších návrhů, jelikož dokáží označit horní a spodní hranice kontrolních kroků. Aritmetický rozdíl těchto hranic vyjadřuje tzv. mobilitu operátoru [3].

Algoritmus **ASAP** plánuje uzly bez předků do prvního kontrolního kroku. Následně vyhledá uzly, jenž mají všechny předky naplánovány a naplánuje je do dalšího kroku. Algoritmus pokračuje, dokud nejsou naplánovány všechny uzly.

Algoritmus **ALAP** plánuje uzly bez potomků do kontrolního kroku očekávaného maxima. Následně vyhledá uzly, jenž mají všechny potomky naplánovány a naplánuje je do předchozího kroku. Algoritmus pokračuje, dokud nejsou naplánovány všechny uzly.

Obrázek 1.3 zobrazuje výsledky základních plánovacích algoritmů ASAP a ALAP s mobilitou operátoru, kde operace byly rozděleny do 4 podgrafů.



Obr. 1.3: Plánování ASAP, ALAP a mobilita operátoru.

Časově omezené plánování

Při časově omezeném plánování se udává maximální počet kontrolních kroků, do kterých algoritmus naplánuje dané operace při co nejmenším možném využití zdrojů. Takové plánování lze řešit exaktním výpočtem pomocí celočíselného lineárního programování, avšak díky své náročnosti na výpočet je tato metoda vhodná pouze pro malé úlohy. Pro větší úlohy se používají metody heuristické.

Force-Directed Heuristic je jedním z heuristických algoritmů pro časově omezené plánování. Cílem je rovnoměrně rozprostřít operátory stejného typu do různých kontrolních kroků a vyvážit očekávanou cenu realizace přes všechny kontrolní kroky [3].

Prostorově omezené plánování

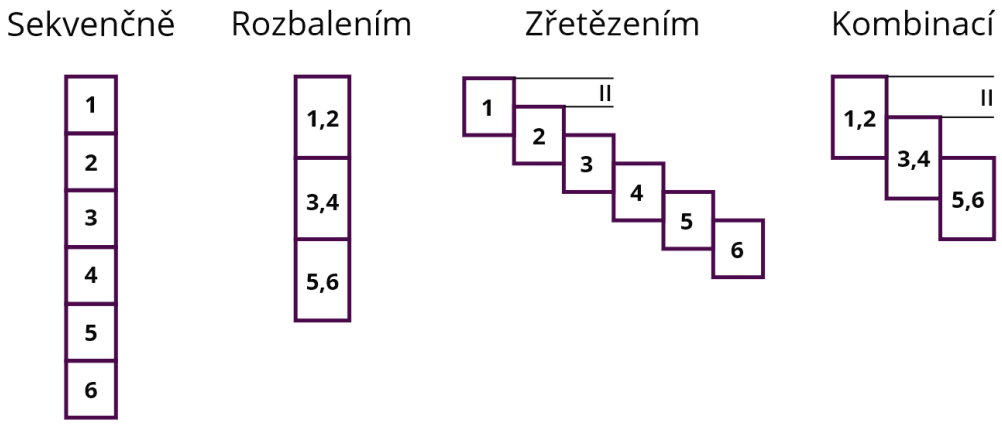
Při prostorově omezeném plánování se udává maximální počet dostupných zdrojů, které algoritmus naplánuje do co nejméně kontrolních kroků. Dostupné zdroje mohou být zadány jako počet funkčních jednotek nebo základních logických hradel.

List-Based Scheduling je algoritmus omezen počtem funkčních jednotek a jde o upravenou variantu algoritmu ASAP. Výstupem algoritmu je seznam uzlů, v kterém jsou uzly uspořádány podle prioritní funkce, na níž závisí kvalita plánování [3]. Možné prioritní funkce jsou podle:

- Mobility – upřednostňovány uzly s nejnižší mobilitou.
- Nejdelší cesty – upřednostňovány uzly s nejdelší cestou ke konci diagramu.
- Počtu následovníku – upřednostňovány uzly s nejvíce následovníky.

Plánování smyček

Plánování smyček má značný vliv na výslednou paralelizaci návrhu, tedy i rychlost. Obrázek 1.4 znázorňuje různé způsoby plánování smyček.



Obr. 1.4: Metody plánování smyček.

V případě **sekvenčního** plánování smyčky je každá iterace zpracována postupně.

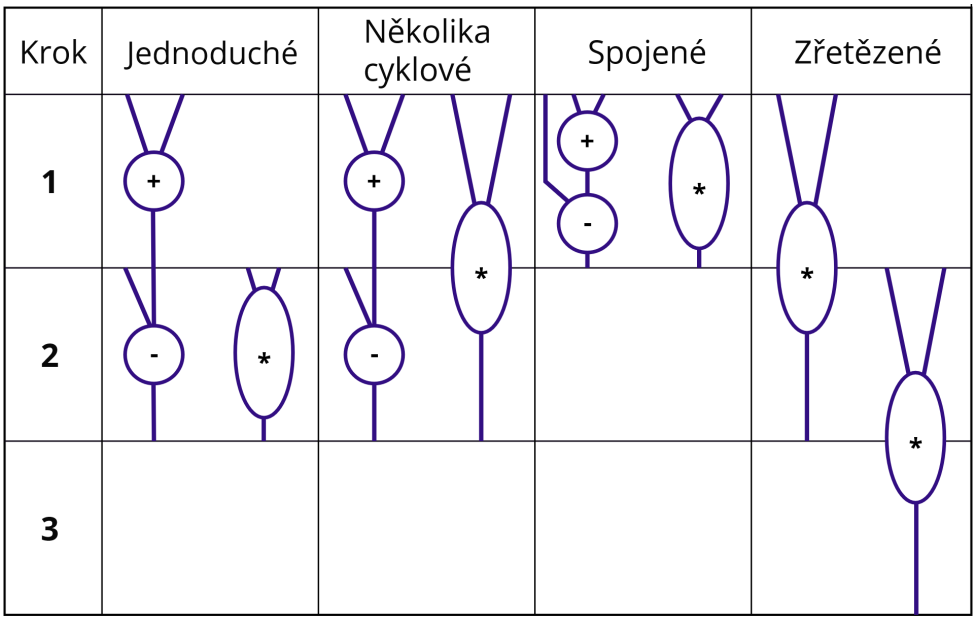
Lepšího řešení plánování s větším smyčkovým paralelizmem je dosaženo **rozbalením** smyček (anglicky loop unrolling) [4], kdy je smyčka rozbalena do tzv. superiterací. Superiterace jsou zpracování sekvenčně a obsahujících určitý počet iterací. Nevýhodou metody je možnost její aplikace pouze u předem známého počtu iterací.

Další metodou je **zřetězení** smyček (anglicky loop pipelining), kdy těla iterací jsou překryta ve smyslu zřetězeného zpracování [3]. Při zřetězení je důležité správně nastavit inicializační interval (anglicky Initialization Interval, zkratkou II) smyčky. II smyčky je počet hodinových cyklů před zpracováním dat další iterace smyčky [5]. Pokud jsou mezi iteracemi vzájemné datové závislosti a II smyčky je nastaven špatně, zřetězení neproběhne, jelikož bude narušena vzájemná datová závislost operací.

Poslední možností je **kombinace metod rozbalení a zřetězení** smyček, kdy se rozbalí smyčka do superiterací, které jsou následně zřetězeny. Opět je aplikace možná jen u předem známého počtu iterací.

Plánování reálných obvodů

Při plánování reálných obvodů je nutné zohlednit i různou délku výpočtu každé operace. Složité operace mohou být provedeny v rámci několika kontrolních kroků, zatímco jednoduché operace mohou být provedeny ve zlomcích kontrolního kroku. Obrázek 1.5 znázorňuje různé situace rozložení operací v kontrolních krocích.



Obr. 1.5: Rozložení operací v kontrolních krocích.

- Jednoduché – každá operace je naplánována do jednoho kontrolního kroku.
- Několika cyklové – některé operace trvají déle, jsou tedy naplánovány do více kontrolních kroků.
- Spojené – některé operace jsou velmi rychlé, jsou tedy naplánovány do jednoho kontrolního kroku.

- Zřetězené – některé operace jsou zřetězeny, aby do nich mohla vstupovat data každý kontrolní krok.

Plánování též uvažuje podmíněné sekce. Výpočet jedné sekce je vyloučen s výpočtem druhé, proto lze sdílet zdroje přímo mezi sekcemi.

1.1.4 Přiřazení zdrojů

Cílem přiřazení zdrojů je mapovat proměnné a operace naplánovaného CDFG do funkčních jednotek, registrů a propojovacích sítí [3]. Tradiční algoritmy přiřazování zdrojů pracují na principu sdílení zdrojů s cílem zmenšit využití prostoru [6]. Typickým příkladem je sdílení zdrojů mezi vzájemně vyloučenými podmíněnými sekcemi.

U funkčních jednotek je nutné přiřadit tu nejlepší kombinaci, pokud je k dispozici více jednotek stejného typu. U paměťových elementů jsou konstanty uloženy do ROM, proměnné do registrů nebo RAM. Registry mohou být sdíleny na základě doby života, tedy počtu kontrolních kroků mezi prvním a posledním použitím proměnné. Propojovací sítě jsou pokud možno sdíleny pro nezávislé přenosy.

Konstruktivní přístup je jednou možností přiřazení zdrojů. Přístup postupně alokuje funkční jednotky. Pokud je více možností, vybere se ta s nejmenší cenou vodičů. Registry jsou přiřazovány podobně. Tento přístup nedává příliš dobré výsledky.

Left-Edge algoritmus je další možností. Každá proměnná má svou dobu života a je mapována do registru. Cílem je namapovat všechny proměnné do co nejmenšího počtu registrů [3]. Algoritmus optimálně využije registry, avšak nebere v úvahu ostatní úlohy alokace.

Dekompozice grafu na kliky je poté vhodnou možností přiřazení zdrojů. U dekompozice jde o rozdělení grafu na minimální počet podgrafů, tedy klik.

V rámci přiřazení operací k funkčním jednotkám se částečně zohledňuje i přiřazení paměťových elementů. Při přiřazování paměťových elementů se také částečně zohledňuje i přiřazení propojovacích sítí.

1.1.5 Alokace zdrojů

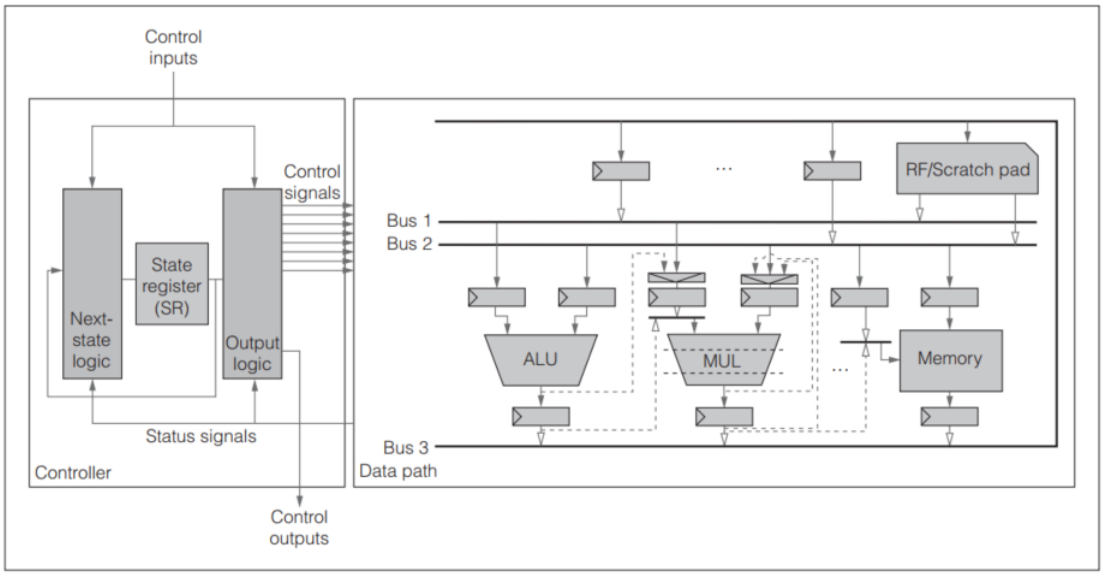
Alokace definuje typ a počet hardwarových prostředků (např. funkční jednotky, paměťové prvky a propojení) potřebné k uspokojení návrhových omezení [1]. Zdroje se přiřazují z knihovny RTL, ve které musí být popsány náležité parametry (např. plocha, zpoždění) potřebné k syntéze. Výběr cílových zdrojů závisí na použité technologii.

Výběr funkčních jednotek se často provádí jako součást plánování. Alokace paměťových elementů (registry, RAM, ROM, FIFO) často probíhá v průběhu přiřazení

proměnných a alokace propojení (různé sběrnice) v průběhu přiřazení propojovací sítě.

1.1.6 Generování RTL

Poté, co byly provedeny předchozích úkoly alokace, plánování a přiřazení, je dalším krokem generování syntetizovaného návrhu se všemi návrhovými rozhodnutími do podoby RTL modelu [1]. Obrázek 1.6 znázorňuje typický příklad výsledné architektury, jenž se skládá z datové cesty a řadiče.



Obr. 1.6: Typická architektura generovaného RTL (převzato z [1]).

Datové vstupy/výstupy jsou propojeny s datovou cestou, kontrolní vstupy/výstupy s řadičem, přičemž v některých architekturách mohou být kontrolní funkce obsaženy v samotné datové cestě.

Datová cesta je složena z funkčních jednotek a jejich propojení. Řadič je reprezentován jako konečný automat řídicí běh datové cesty pomocí kontrolních vstupů a stavových signálů. Řadič obsahuje:

- Stavový registr – uchovávající momentální stav, který je zároveň stavem konečného automatu.
- Logiku dalšího stavu – počítající další stav k načtení.
- Výstupní logiku – generující kontrolní signály a výstupy.

Mapování kroků kontrolních stavů na stavy automatu může být lokálního rozdělení, kde se u větvení grafu na vzájemně vyloučené části generují stavy ke každé části zvlášť. Druhou možností je globální rozdělení, kde se u větvení grafu na vzájemně vyloučené části generují stavy pro obě části společně, avšak je potřeba stavového

registru k rozlišení právě vykonávající se větve. Globální rozdělení poskytuje lepší optimalizaci a výsledná využitá plocha je menší.

1.2 Jazyk P4

Programming Protocol-Independent Packet Processors (dále jen P4) je programovací jazyk, který vyjadřuje, jak jsou pakety zpracovávány datovou rovinnou programovatelného přeposílacího prvku, například hardwarového nebo softwarového přepínače, karty síťového rozhraní, směrovače nebo síťového zařízení [7]. Oproti předchůdcům (např. OpenFlow) dovoluje jazyk P4 definovat vlastní protokoly. Právě neustále nové protokoly jsou důvodem otevřeného návrhu, který dodává větší flexibilitu. Tato část práce vychází především z [7, 8, 9, 10].

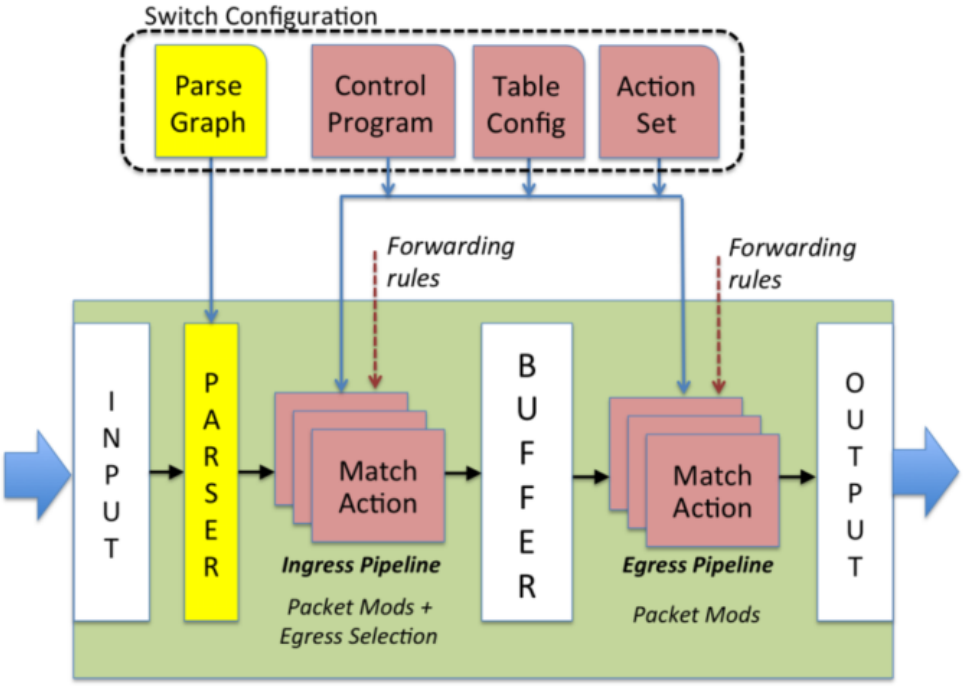
Zaměření jazyka P4

- **Nezávislost platformy**. P4 program je nezávislý na cílové výpočetní jednotce zvané platforma P4, tedy může být kompilován pro jakoukoliv z nich. Platformou P4 může být CPU, FPGA a jiné. Kompilátor uvažuje možnosti síťového přepínače až při převodu platformě-nezávislého popisu (napsaného v jazyce P4) na platformě-závislý program (použitý pro konfiguraci síťového přepínače).
- **Nezávislost protokolů**. Jazyk nemá žádnou základní podporu protokolů. Ty jsou specifikovány až samotným programátorem a zpracovány kompilátorem pro danou platformu P4.
- **Rekonfigurovatelnost**. Platformy P4 by měly být schopné změny zpracování paketů (např. parsování, zpracování hlaviček) i poté, co byly uvedeny do provozu.

1.2.1 Abstraktní model přeposílání

V abstraktním modelu přeposílání na obrázku 1.7, přeposílají síťové přepínače pakety skrz programovatelný parser, následovaný několika úseky match+action tabulek uspořádaných sériově, paralelně, nebo jako kombinace obou [8].

Nejprve dojde k zpracování paketů v parseru, jenž extrahuje pole hlaviček a definuje podporovaný protokol. Následně jsou extrahované hlavičky posunuty do match+action tabulek rozdělených na vstupní a výstupní tabulky, v kterých mohou být hlavičky modifikovány. Vstupní tabulky definují výstupní port a frontu, do které je paket zařazen. Výstupní tabulky provádějí modifikace jednotlivých hlaviček paketů (např. multicastové kopie). Metadata, tedy dodatečné informace mezi jednotlivými stádii zpracování paketů, jsou zpracována stejně jako hlavičky.



Obr. 1.7: Abstraktní model přeposílání (převzato z [8]).

1.2.2 Konstrukce jazyka P4

Na základě abstraktního modelu přeposílání byl vytvořen jazyk, který popisuje jak má být síťový přepínač nakonfigurován, a jak se mají pakety zpracovávat [8]. P4 program se skládá z těchto základních konstrukcí:

- **Formáty hlaviček** popisují hlavičky protokolů.
- **Parser paketů** popisuje povolené sekvence hlaviček v přijatých paketech, jak identifikovat tyto sekvence hlaviček a extrahuje hlaviček a pole z paketů.
- **Specifikace tabulek** popisují stav, který slouží k předávání paketů. Obecně jde o tradiční tabulky síťového přepínače.
- **Specifikace akcí** popisují, co je provedeno s pakety a metadaty dané hlavičky.
- **Jednotka match+action** spojuje funkci tabulek a akcí.
- **Definice řízení toku** popisuje pořadí, ve kterém jsou match+action tabulky vykonány.
- **Externí objekty** jsou speciální architektury nepopsatelné samotným jazykem P4, avšak zařazeny do P4 programu (pouze P4 16).
- **Uživatelská metadata** jsou uživatelem definovaná data spojená s paketem.
- **Intrinsická metadata** jsou data spojená s architekturou paketu (např. vstupní port, kde má být paket zpracován).

Příklady základních konstrukcí

Formát hlavičky je specifikován seznamem polí hlaviček paketů a jejich bitovou šířkou. Jazyk P4 používá jednoduchý syntax ve formě *jméno : šířka* [9]. Příklad formátu hlavičky:

Výpis 1.1: Příklad hlavičky (převzato z [8]).

```
1 header ethernet {
2     fields {
3         dst_addr : 48; // width in bits
4         src_addr : 48;
5         ethertype : 16;
6     }
7 }
```

Parser paketů zajišťuje projití hlaviček paketů od začátku do konce, přičemž postupně extrahuje hodnoty polí následně využitých v match+action tabulkách. Tuto funkci zajišťuje stavový automat, který provádí přechody mezi hlavičkami. Příklad parseru paketů:

Výpis 1.2: Příklad parseru paketů (převzato z [9]).

```
1  header ethernet eth;
2  parser start {
3      return parse_ethernet;
4  }
5
6  parser parse_ethernet {
7      extract(eth);
8      switch(eth.ethertype) {
9          case 0x8100: vlan;
10         case 0x9100: vlan;
11         case 0x0800: ipv4;
12         case 0xA100 mask 0xF100 : myProto;
13         default : ingress;
14     }
15 }
```

Příkaz *parser start* zahajuje parsování, dokud není ukončeno chybou či příkazem *stop*. Při použití příkazu *extract* dojde k prozkoumání a extrakci vstupních paketů a dat definovaných v hlavičce. Příkaz *switch* poté rozhoduje o dalším stavu parsovaných dat. Příkaz *mask* provádí nad hlavičkou logickou operaci *and* mezi hodnotou a maskou. Výsledek je porovnán s hodnotou *0xA10*.

Specifikace tabulek popisuje, jak by měla být extrahovaná pole hlaviček co namapována na vhodné akce. Příklad match+action tabulky:

Výpis 1.3: Příklad match+action tabulky (převzato z [9]).

```
able filter {
    reads {
        ipv4.srcAddr : lpm;
        tcp.dstPort : exact;
    }
    actions {
        PushVlan;
        Permit;
        NoOp;
    }
}
```

Příkaz *reads* určuje způsob vyhledávání v tabulce a je kvalifikován typem shody. V tomto případě jsou typy shody *exact*, kdy hodnota pole musí být stejná jako záznam v tabulce a *lpm*, kdy je hledán nejdelší binární prefix. Příkaz *actions* poté obsahuje podporované akce, které mohou obsahovat parametry, jenž jsou uloženy v match+action tabulkách [9].

Specifikace akcí se definuje ze seznamu primitivních akcí (více ve specifikaci jazyka [10]), z kterých se dají vytvořit akce komplikovanější. Každá akce může volat akci jinou. Akce jsou vykonávány sekvenčně a mohou přijímat některé parametry, jenž mohou být změněny během chodu. Příklad definice akcí:

Výpis 1.4: Příklad specifikace akcí (převzato z [8]).

```
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set the VLAN's ethertype to signal mTag
    copy_field(vlan.ethertype,0xaaaa);
    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);
    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

V příkladu je mTag header vložen za VLAN header. Je zkopírován ethertype z VLAN headeru do mTagu a ethertype z VLAN je nastaven na 0xaaa, aby bylo signalizováno použití mTag protokolu.

Externí objekt popisuje sadu metod, jenž jsou implementovány objektem, ale nepopisuje implementaci těchto metod (tedy je podobný abstraktní třídě v objektově orientovaném jazyku) [7]. Příklad definice externího objektu:

Výpis 1.5: Příklad definice externího objektu (převzato z [7]).

```
externChecksum16 {
    Checksum16();
    voidclear();
    voidupdate<T>(inT data);
    voidremove<T>(inT data);
    bit<16> get();
}
```

Na příkladu kontrolního součtu je nejprve zavolán konstruktor a následně je jednotka připravena pro výpočet. Poté jsou přidána data ke kontrolnímu součtu, odstraněna data z existujícího kontrolního součtu a nakonec jsou získána data kontrolního součtu, jenž byla přidána od posledního vyčištění.

Definice řízení toku se provádí pomocí tzv. control flow, jenž je specifikována jako set funkcí, podmínek a odkazů na tabulky. Příklad řízení toku:

Výpis 1.6: Příklad řízení toku (převzato z [10]).

```
control main() {
    // Verify mTag state and port are consistent
    table(source_check);
    // If no error from source_check, continue
    if (!defined(metadata.ingress_error)) {
        // Attempt to switch to end hosts
        table(local_switching);

        if (!defined(metadata.egress_spec)) {
            // Not a known local host; try mtagging
        table(mTag_table);
        }

        // Check for unknown egress state or
        // bad retagging with mTag.
        table(egress_check);
    }
}
```

V příkladu řízení toku nejprve dojte k verifikaci konzistence mezi přijatými pakety a vstupním portem pomocí tabulky *source_check*, která zbaví paket mTagu. Pokud vše proběhne správně, přejde se k tabulce *local_switching*. Pokud se nestrefí, nejde o lokálně připojeného hosta a přejde se k tabulce *mTag_table*, kde dojde k přidání mTagu (viz výpis 1.4). *Egress_check* tabulka se nakonec stará o přeposílání. U neznámých cílů posílá upozornění kontrolnímu zásobníku.

1.2.3 Kompilace P4 programu

U P4 programu je nutné namapovat platformě-nezávislý popis na platformě-závislou hardwarovou nebo softwarovou platformu. V případě programovatelného parseru je parser kompilován do stavového automatu, zatímco v případě fixovaného parseru dojde k pouhé verifikaci kompatibility s platformou. V případě řídicího toku dojde k nalezení závislostí hlaviček a také tabulek, tedy k jejich možnému paralelnímu zpracování.

Kompilátor následuje dvouúrovňový proces. Nejprve konvertuje řídicí program P4 na přechodnou reprezentaci (diagram závislosti tabulek), kterou analyzuje pro určení závislosti mezi tabulkami. Platformě specifický back-end (např. P4-to-VHDL) následně namapuje diagram na cílové zdroje přepínače.

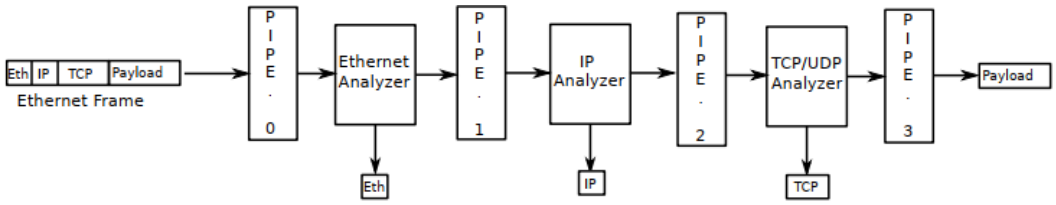
1.3 Kompilátor P4-to-VHDL

Kompilátor P4-to-VHDL převádí platformě-nezávislý program napsaný v jazyce P4 na platformě-závislý program popsaný v jazyce VHDL. Platformou P4 je tedy síťové zařízení s FPGA, které se v poslední době stalo velmi rozšířeným nejen pro síťové aplikace jako je SmartNIC. P4-to-VHDL je vytvořeno jako sada skriptů v jazyce Python a je vyvíjeno v rámci sdružení CESNET. Původní verze pracuje s P4 14 a přechází se na verzi P4 16, pro kterou je vytvářen nový kompilátor kvůli zpětné nekompatibilitě P4 14 s P4 16.

Základní komponenty vytvořené v P4-to-VHDL

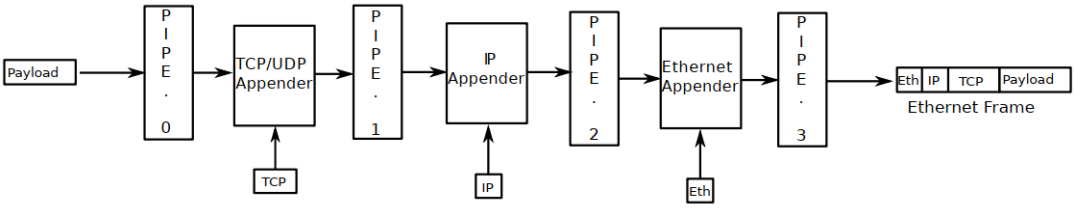
Popis převodu jednotlivých komponent z jazyku P4 na VHDL je nad rámec této práce, avšak detailní popis může být nalezen v [9, 11]. V rámci kompilátoru P4-to-VHDL jsou vytvořeny následující komponenty.

Parser je založen na architektuře HFE M2 (obrázek 1.8) složené ze dvou hlavních bloků – analyzátoru protokolů a pipeliny [11]. Parser obsahuje volitelnou pipeline pro doladění frekvence, propustnosti a využitého prostoru.



Obr. 1.8: Architektura HFE M2 [9].

Deparser je založen na architektuře podobné HFE M2 (obrázek 1.9) složené z protokolových appenderů a pipeline [9]. Tak jako u parseru, obsahuje deparser volitelnou pipeline. Deparser je založen na méně efektivním, obecném vkládání hlavičky před přenášená data (anglicky payload).



Obr. 1.9: Architektura deparseru [9].

Match+action sekce jsou sestaveny v závislosti na požadované funkcionalitě. Momentálně umožňují modifikovat hodnoty polí protokolů, vkládat/odebírat hlavičky. Podporují metadata a tabulky je možné plnit při běhu. Jsou podporovány vyhledávací algoritmy jako TCAM, TCAM a Cuckoo Hashing (pouze exact shoda).

Způsob překladu kódu do návrhu

Způsob překladu P4 kódu do návrhu probíhá následujícím způsobem:

1. P4 kód se předá P4-to-VHDL překladači, který vygeneruje VHDL kód P4 pipeliny a stromovou struktura dat popisující hardwarové komponenty (anglicky Device Tree).
2. Tento generovaný kód se předá Tcl souboru, jenž doplní hodnoty nezjistitelné při překladu.
3. Spustí se syntéza a daný Device Tree (včetně logické struktury P4 zařízení) se nahraje do komponenty v kartě. Poté bude knihovna P4 zařízení schopna složit konfigurační tok do karty přes rozhraní MI32.

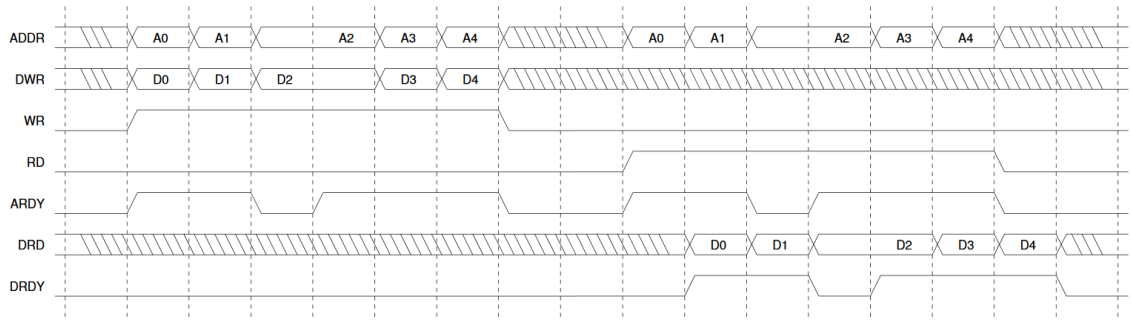
Paměťové rozhraní MI32

Paměťové rozhraní MI32 (anglicky Memory Interface) propojuje software s hardware a slouží k zápisu a čtení z paměti po 32 bitech. Rozhraní obsahuje signály uvedené v tabulce 1.1.

Tab. 1.1: Signály paměťového rozhraní MI32.

Název	Typ	Šířka [bit]	Význam
DWR	Vstup	32	Data pro zápis.
ADDR	Vstup	32	Adresa zápisu.
RD	Vstup	1	Čtecí požadavek.
WR	Vstup	1	Zápisový požadavek.
BE	Vstup	4	Signál pro určení platnosti jen některých bitů dat.
DRD	Výstup	32	Data k načtení.
ARDY	Výstup	1	Potvrzení přijetí adresy.
DRDY	Výstup	1	Indikace platnosti načtených dat.

Při zápisové transakci je nejprve nastaven signál ADDR s adresou zápisu. Přijetí adresy je potvrzeno signálem ARDY. Při nastavení signálu WR začíná zápis dat signálu DWR na danou adresu. Čtecí transakce začíná stejně jako transakce zápisová, je nastaven signál ADDR s adresou čtení a signál ARDY potvrzuje přijetí adresy. Při nastavením signálu RD začíná čtení, v kterém jsou po určité době přijímána vyčtená data signálu DRD z dané adresy a přijetí dat je potvrzeno signálem DRDY. Na obrázku 1.10 lze vidět nejprve zápisovou, poté čtecí transakci.



Obr. 1.10: Zápisová a čtecí transakce paměťového rozhraní MI32 (převzato z [13]).

1.3.1 Propojení vysokoúrovňové syntézy

Programovací jazyk P4 si lze představit jakožto ideální platformě-nezávislý programovací jazyk pro popis síťových prvků. Zajišťuje to jeho specializace, která vytváří potřebnou míru abstrakce. Platformě-závislým programovacím jazykem poté může být VHDL, C++ a jiné.

VHDL má oproti P4 velmi malou úroveň abstrakce, což je způsobeno jeho univerzálností a schopnosti přesně popsat návrh. To však vede k jeho nevýhodě, kterou je časová náročnost návrhu a s tím spojené náklady.

HLS (C, C++ a SystemC) tedy stojí někde uprostřed a nabízí vysokou míru abstrakce, dostatečnou univerzálnost a rychlost popisu. Automatické optimalizační techniky poté zaručí nejlepší výsledky návrhu.

Použití HLS v rámci P4 lze vidět ve článku [14]. Článek se zabývá návrhem parseru paketů s pomocí jazyka P4, který využívá popisu v C++, z něhož je vygenerováno RTL skrze Vivado HLS. Výsledný návrh dosahuje vyšších rychlostí a nižší latence, než návrh vygenerovaný z jazyka VHDL (parser v rámci P4-to-VHDL). To je způsobeno právě optimalizačními schopnostmi HLS nástrojů, převážně automatickým zřetězením a proregistrováním. Nejen proto se jiné HLS nástroje stávají vhodným doplňkem pro P4.

Propojení HLS a P4 dává smysl hlavně v případě jeho nové verze P4.16. Ta, oproti původní verzi P4.14, umožňuje deklarace atomických sekcí, jakékoliv výrazy složené z aritmeticko-logických operací, podporuje podmíněné výrazy a dovoluje zakomponování externích bloků.

Propojení P4 a HLS nabízí, oproti VHDL, hned několik výhod:

- Vysoká míra abstrakce
- Jednoduchý popis návrhu
- Lepší udržitelnost kódu
- Optimalizační schopnosti na úrovní HLS nástrojů
- Rychlost návrhu pramenící z předešlého

1.4 Specifika kompilátoru Intel HLS

Popisu číslicového návrhu v C, či C++ je oproti softwarovému programování trochu odlišný. Popis obsahuje specifické funkce, třídy, pragmy a jiné příkazy (dále konstrukce). Navíc každý kompilátor může, a také většinou přistupuje, k problematice syntézy trochu jiným způsobem. V této kapitole jsou popsána specifika kompilátoru Intel HLS (dále jen Intel HLS) verze 19.4, která jsou v rámci této práce použita. Tato část práce vychází především z [15, 16].

1.4.1 Proces návrhu komponenty

Proces návrhu komponenty v Intel HLS probíhá následujícím způsobem:

1. **Popis komponenty a testbench** v jazyce C a C++.
2. **Kompilace a softwarová simulace** slouží jako funkční verifikace.
3. **Syntéza a ko-simulace** generuje IP (Intellectual Property) core a provádí verifikaci návrhu skrze ModelSim.
4. **Syntéza RTL do hradel cílové technologie** pomocí Quartus Prime, generuje konfigurační bitstream a QoR (Quality of Results).
5. **Integrace do systému** skrze Quartus Prime nebo Platform Designer.

Popis komponenty a testbench se běžně provádí v rámci jednoho souboru. Funkce komponenty je označena klíčovým slovem *component* a popisuje svou funkcionalitu. Testbench je vždy jen jeden, označen jako funkce *main*. Začátek testbench obsahuje množinu testovacích vektorů, kterými je komponenta testována a v závěru jsou načítána výstupní data komponenty.

Generované reporty

Po kompilaci komponety generuje Intel HLS skupiny reportů, které umožňují analyzovat různé aspekty komponenty jako zdroje, struktury smyček, použití pamětí a zřetězení komponenty [15]. Reporty se zobrazují skrze soubor *reports.html* otevřený v prohlížeči a skládají se ze čtyř hlavních sekcí:

- Souhrn (Summary) – obsahuje základní informace (např. verze kompilátoru, příkaz kompilace, cílové zařízení), syntetizované funkce (komponenty), QoR, předpokládané využití zdrojů a varování kompilátoru.
- Analýza propustnosti (Throughtput Analysis) – obsahuje výsledky II a latence jak celé komponenty, tak jednotlivých funkcí a smyček. Také obsahuje informace, zdali byla komponenta zřetězena.
- Analýza využití zdrojů (Area Analysis) – obsahuje podrobné výsledky využití zdrojů.
- Prohlížeč systému (System Viewer) – obsahuje informativní diagram celého systému, pamětí a obsahuje prohlížeč plánování.

1.4.2 Specifické konstrukce

Pro správný popis komponenty obsahuje kompilátor specifické konstrukce popisující rozhraní komponenty, například datové typy s bitovou přesností, různé volání funkcí a konstrukce umožňující optimalizaci smyček, pamětí či konstrukce umožňující práci s datovými typy.

Volání funkcí

Klíčové slovo *component* označuje jednu funkci a její podfunkce jako komponentu. Uvnitř této funkce jsou přímo volané podfunkce inlinovány, zatímco podfunkce volané skrze tzv. systém úloh, anglicky System of Tasks API (Application Program Interface), generují hardware mimo datovou cestu komponenty a chovají se jako asynchronní volání [16]. Díky systému úloh je možné vykonávat smyčky paralelně, sdílet drahé výpočetní bloky, či strukturovat návrh hierarchicky. Výpis 1.7 popisuje způsoby volání funkcí.

Výpis 1.7: Příklad volání funkcí.

```
1  void subfuncion() {
2      // Anything
3  }
4
5  componet void funcion() {
6      // Synchronous call
7      subfuncion();
8      // Asynchronous calls
9      ihc::launch(subfuncion);
10     ihc::collect(subfunction);
11     ihc::launch_always_run<subfunction>();
12 }
```

Příkaz *ihc::launch* označuje funkci jako úlohu pro hardwarovou generaci a spouští úlohu asynchronně, *ihc::collect* poté synchronizuje ukončení specifické úlohy komponenty. Příkaz ihc:: *launch_always_run* označuje funkci jako úlohu, kterou spouští okamžitě po zapnutí komponenty a úloha je poté vykonává neustále.

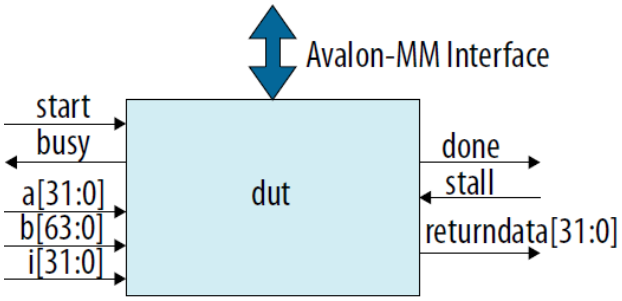
Rozhraní

Aby mohla být komponenta zapojena do většího celku, musí mít odpovídající rozhraní. Intel HLS rozlišuje mezi tzv. rozhraním pro vyvolání komponenty a parametrovým rozhraním.

Pro každou funkci označenou příkazem *component* (komponenta), vytváří Intel HLS odpovídající RTL modul. Tento RTL modul musí mít tzv. top-level porty, nebo rozhraní, které umožňuje okolnímu systému interagovat s HLS komponentou [16]. Rozhraní pro vyvolání komponenty vždy obsahuje signály *start*, *busy*, *done*, *stall*, v případě návratové hodnoty obsahuje i signál *return*. Výpis 1.8 popisuje jednoduchou komponentu s návratovou hodnotou a obrázek 1.11 vyobrazuje její vytvořené rozhraní.

Výpis 1.8: Příklad základního rozhraní (převzato z [16]).

```
component int dut(int a, int* b, int i) {
    return a*b[i];
}
```



Obr. 1.11: Základní rozhraní funkce s návratovou hodnotou (převzato z [16]).

Jedním z parametrových rozhraní je rozhraní stream (Avalon Streaming Interface), které je implementováno jako paměť FIFO s implicitními signály *valid* a *ready*. Ostatní parametry a signály mohou být explicitně nastaveny. V běžném případě se rozlišuje mezi vstupním a výstupním rozhraním stream. V případě propojení dvou funkcí interní FIFO pamětí se používá rozhraní stream s oběma konci. Přístup k rozhraní může být blokující či neblokující. V případně blokujícího volání se komponenta pozastaví, pokud je paměť FIFO plná či prázdná. V případě neblokujícího volání je navrácen Boolean, který rozhoduje o úspěšnosti volání. Výpis 1.9 popisuje zápis rozhraní stream na globální úrovni a jeho následné volání na úrovni funkce.

Výpis 1.9: Příklad rozhraní stream a jeho volání.

```
// Set streaming interface (global)
ihc::stream_in<int> data_in;
ihc::stream_out<int> data_out;
ihc::stream<int> internal_connection;

// Call streaming interface (inside function)
int var = data_in.read();             // Blocking read
int var = data_in.tryRead(success); // Non-blocking read
data_out.write(var);                  // Blocking write
success = data_out.tryWrite(var);    // Non-blocking write
```

V rámci simulace je každé explicitní volání komponenty blokující a testbench čeká na návratovou hodnotu komponenty před dalším zpracováním, což ústí v sériové provedení [15]. Pro explicitní rozhraní stream mohou být signály zařazeny do fronty pomocí speciální funkce, kterou popisuje výpis 1.10.

Výpis 1.10: Příklad zařazení testovacích dat do fronty komponenty.

```
1 // With return value and internal inputs
2 ihc_hls_enqueue(&result, &top, var1, var2);
3 // Without return value and external inputs
4 ihc_hls_enqueue_noret(top);
5 // Execution of enqueued component calls
6 ihc_hls_component_run_all(top)
```

Příkaz *ihc_hls_enqueue* vyvolává jedno zařazení do fronty komponenty s návratovou hodnotou. Příkaz *ihc_hls_enqueue_noret* provede to samé pro komponentu bez návratové hodnoty. Příkaz *ihc_hls_component_run_all* následně tato zařazení vykoná.

Datové typy

Pro návrh hardware je potřeba datových typů s bitovou přesností, aby nebyly spotřebovány nadbytečné zdroje. Intel HLS k tomu využívá knihoven od firmy Mentor Graphics.

Knihovny poskytují numerické C++ třídy a rozhraní, které je zaměřeno na modelování vlastností určených pro návrh hardware [17]. Numerický balíček poskytuje třídy pro bitově přesná celá čísla (bit-accurate integer), pevnou řádovou čárku (fixed-point), plovoucí řádovou čárku (floating-point) a komplexní čísla (complex numbers).

Výpis 1.11 popisuje příkazy použité v rámci této práce. Příkazy jsou z knihovny ac_int.h, jenž umožňuje práci s bitově přesnými čísly.

Výpis 1.11: Příklad práce s datovými typy.

```
1 ac_int<W, S>         // W is Width, S is Signedness
2 slc<W2>(int_type i) // int_type can be ac_int, unsigned, int
3 set_slc(int_type i, ac_int<W2,S2> x)
```

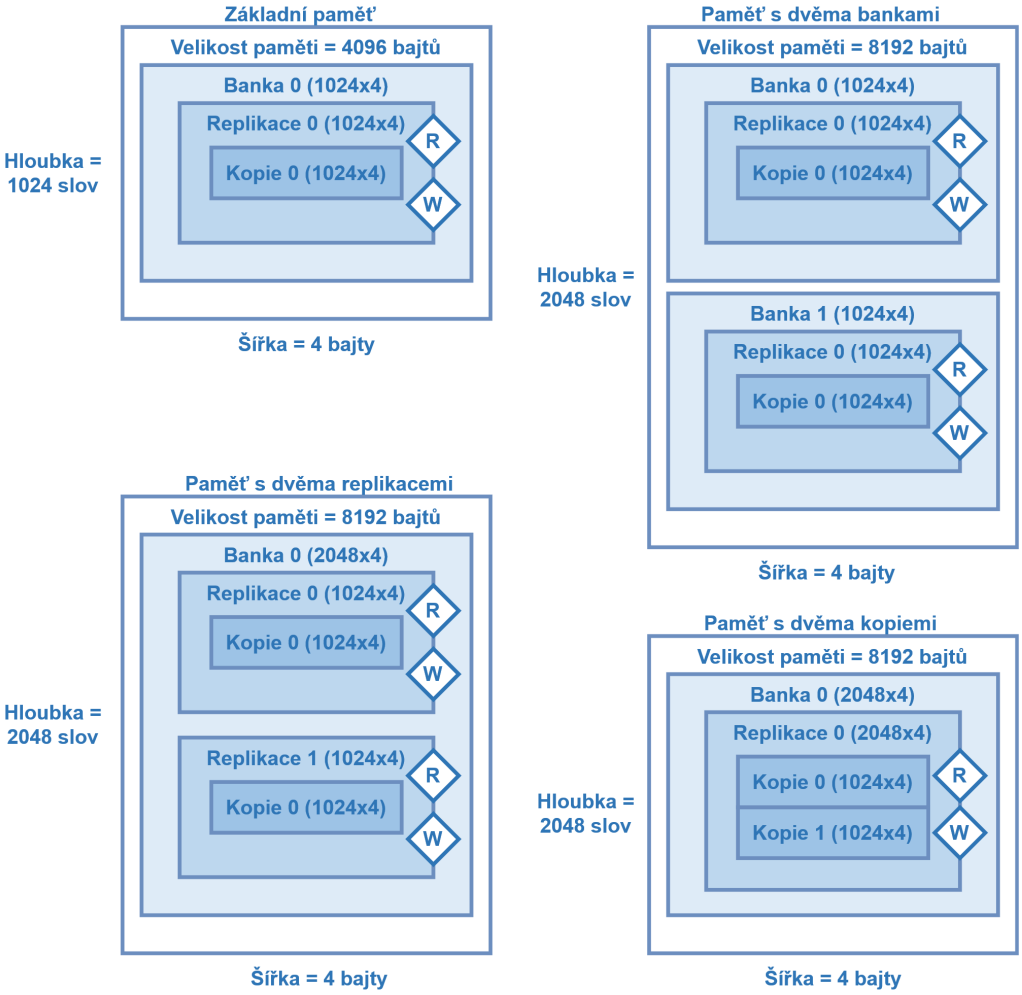
Příkaz *ac_int* specifikuje celé číslo a zdali je signed (*true*), či unsigned (*false*). Příkaz *slc* navrací řez o šířce $W2$ začínající od indexu i (tedy řeže $W2-1+i$ downto i). Příkaz *set_slc* kopíruje bity x do řezu s nejméně významným bitem i (tedy bity $W2-1+i$ downto i jsou nastaveny bity x).

Paměti

Intel HLS vytváří hardwarové paměťové prvky (např. blokové RAM) pro každou *local*, *constant*, *static* proměnnou a pro každé pole [16]. Přístup paměti je namapován do LSU (Load-Store Unit), které přistupují k hardwarové paměti skrze její porty. Systém paměti je následně rozdělen na několik částí:

- **Banky** – data mohou být rozdělena do jedné či více bank, kde každá banka obsahuje určitou podmnožinu dat zapsaných v hardwarové paměti.
- **Replikace** – banka může obsahovat jednu či více replikací, kde každá replikace obsahuje identická data. Replikace umožňuje mít dva (čtyři) zápisové a čtecí porty v rámci jedné banky pracující na stejné (dvojnásobné) frekvenci komponenty.
- **Kopie** – replikace obsahuje privátní kopie, kde každá kopie obsahuje identická data a umožňuje práci se souběžnými smyčkami.

Základní konfigurace paměti má hloubku 1024 slov o šířce 4 bajtů, velikost paměti je tedy 4096 bajtů. Obrázek 1.12 zobrazuje základní konfiguraci paměti, paměť s dvěma bankami, dvěma replikacemi a dvěma kopiemi.



Obr. 1.12: Konfigurace paměti.

Intel HLS automaticky optimalizuje konfiguraci paměti. Konfigurace může být případně usměrněna atributy, které kompilátoru nařídí specifickou implementaci pa-

měti. Výpis 1.12 popisuje atributy použité v rámci této práce.

Výpis 1.12: Příklad optimalizačních atribut paměti.

```
1 hls_memory_impl()          // Memory implementation
2 hls_max_replicates()       // Number of memory replications
3 hls_numbanks()             // Number of memory banks
4 hls_bankwidth()            // Width of memory banks in bytes
```

První příkaz specifikuje implementaci paměti (např. bloková RAM, registry). Druhý příkaz specifikuje počet kopií paměti, které určují počet přístupů do paměti. Třetí příkaz specifikuje z kolika bank se paměť skládá. Čtvrtý příkaz specifikuje o šířce paměti.

2 Praktická část

V rámci praktické části je nejprve popsán koncept návrhu akcí, externích bloků a přístupu MI32. Následně je popsána funkcionalita jednotlivých komponent a v závěru jsou zhodnoceny dosažené výsledky.

2.1 Koncept akcí, externích bloků a přístupu MI32

Cílem konceptu je navržení a propojení akcí s externími bloky, které prioritně komunikují se software skrze rozhraní MI32. V rámci konceptu jsou akce chápány jako část systému, která provádí různé úkony (např. operace se vstupními hodnotami či registry) na základě hlaviček paketů a parametrů akce. Externími bloky se chápou právě registry, které jsou akcemi využívány, avšak nejsou jejich přímou součástí. Externí bloky se tedy dají využít i v rámci jakéhokoliv jiného systému. Přístupem MI32 je chápáno napojení řídicí roviny externích bloků na rozhraní MI32.

Požadavky návrhu

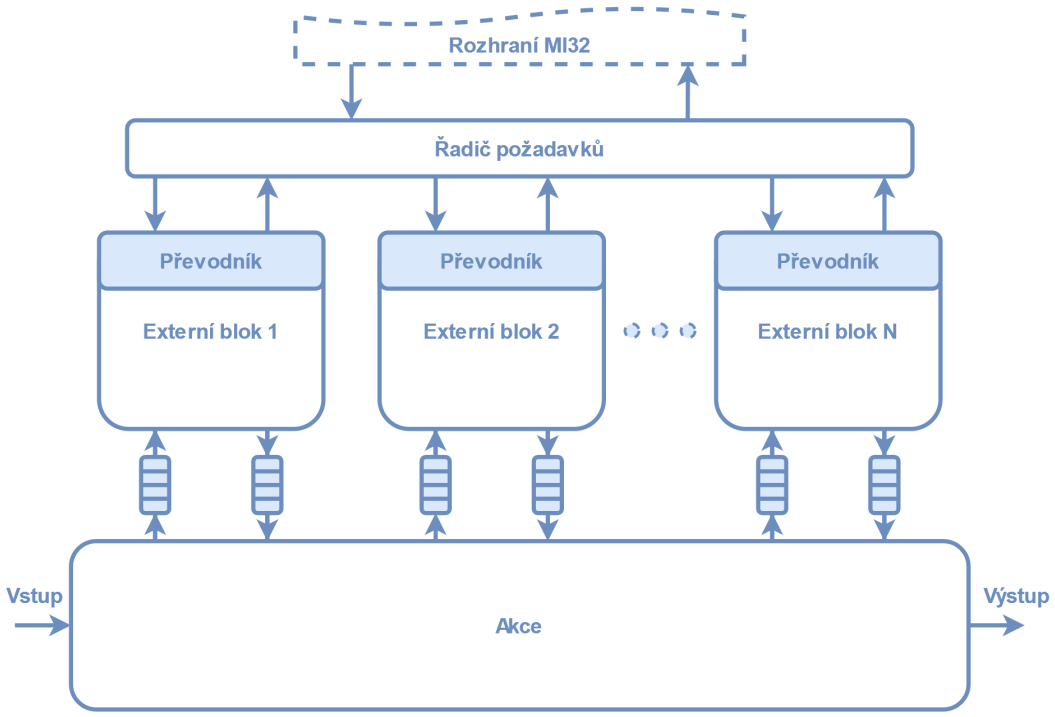
Na návrh je kladeno několik obecných požadavků:

- Priorita kontrolní roviny před datovou rovinou – požadavky ze software musí být odbaveny před požadavky akcí.
- Nezávislost – komponenty jako akce, registry a přístup k nim by měly pracovat nezávisle na sobě, tedy každá komponenta zapisuje a načítá data svou vlastní rychlostí.
- Spolehlivost – v komponentách by nemělo docházet k uváznutí neboli vzájemnému čekání, kdy jedna akce čeká na dokončení druhé a naopak.
- Co nejmenší inicializační interval – II by měl být ve většině případů roven jedné. Výjimkou mohou být případy, kdy komponenta úmyslně čte či zapisuje po určitém počtu taktů hodinového signálu.
- Generický popis – každá komponenta by měla být v ideálním případě nastavena několika parametry. Tím se zajistí jednoduchá implementace komponenty v rámci kompilátoru P4.

Koncept návrhu

Obrázek 2.1 zobrazuje koncept návrhu akcí, externích bloků a přístupu MI32. V horní části obrázku přistupuje rozhraní MI32 k řadiči požadavků, který rozesílá požadavky do správného externího bloku. V případě čtecího požadavku řadič následně čeká na odpověď externího bloku, kterou poté posílá do MI32 rozhraní. V rámci externího bloku je nejprve převodník, který převádí příchozí požadavky datové šířky

32 bitů z rozhraní MI32 na požadavky datové šířky daného registru externího bloku. V případě odpovědi je převod datové šířky reverzní. V dolní části obrázku vstupuje požadavek datové roviny do akcí, které následně přistupují k daným externím blokům. Požadavky akcí a odpovědi externích bloků jsou seřazovány do bufferu, kvůli případnému dlouhému blokování ze strany MI32 rozhraní. Akce na konci navrací výstup datové rovině.



Obr. 2.1: Koncept akcí, externích bloků a přístupu MI32.

2.2 Navržené komponenty

V rámci návrhu bylo vytvořeno několik komponent, které jsou zařazeny do tří hlavních problematik. Nejprve jde o samotnou práci s registry, které jsou součástí externího bloku. Dále jde o návrh nezávisle pracujících akcí a externích bloku. Nakonec jde o napojení rozhraní MI32 na externí bloky.

2.2.1 Práce s registry

Problematika práce s registry se zabývá přístupem k registrům/pamětím (dále jen registrům), jejím správným nastavením a správnou implementací. Nejprve je popsána šablona třídy pro práci s registry. Poté je popsáno základní rozhraní pro přístup

k registrům, které zahrnuje přístup kontrolní i datové roviny k jednomu registru a využívá šablony třídy pro práci s registry. Následné rozhraní pro přístup k více registrům využívá více registrů, k nimž je přístup řešen skrze adresový dekodér. Na závěr je popsána problematika nesprávné implementace přístupu k registrům, jenž odhalila chybu kompilátoru.

Šablona třídy pro práci s registry

Práce s registry je řešena skrze šablonu třídy *Register* v hlavičkovém souboru. Skrze šablonu se předává datový typ, počet položek a šířka adresy registrů. Při následné instanci třídy je v konstruktoru předán ukazatel na paměť. Univerzální třída poté poskytuje metody práce s pamětí, tedy čtení a zápis.

Základní rozhraní pro přístup k registrům

Základní rozhraní pro přístup k registrům obsahuje jeden registr a je rozděleno na datovou a řídicí rovinu. Registr prioritně zpracuje čtecí či zápisové požadavky řídicí roviny, následně datové roviny.

V případě vytvořeného návrhu obsahuje datová rovina jednu akci, která nejprve načítá data ze vstupu, tedy adresu registru. Pakliže jsou data ze vstupu načtena, provede se čtení na dané adrese registru, přečtená data registru se modifikují a zapíší zpět do registru. Nakonec se zapíšou modifikovaná data na výstup akce.

Rozhraní pro přístup k více registrům

Rozhraní pro přístup k více registrům obsahuje více registrů a je rozděleno na řídicí a datovou rovinu stejně jako v případě základního rozhraní pro přístup k registrům. Registry prioritně zpracují čtecí a zápisové požadavky řídicí roviny, následně datové roviny. Adresový dekodér rozhoduje o registru, jenž vykoná daný požadavek.

V případě vytvořeného návrhu jsou v rámci akcí nejprve načtena data ze vstupu, tedy identifikátor akce, adresa registru a vstupní data. První akce sčítá vstupní data a zapisuje výsledek do prvního registru. Druhá akce násobí vstupní data a zapisuje výsledek do druhého registru. Výsledek každé akce je zapsán na výstup v podobě identifikátoru akce a výstupních dat.

Problematika přístupu k registrům

V rámci předešlého návrhu byla zjištěna nesprávná implementace přístupu k registrům, jenž je způsobena kompilátorem Intel HLS. Pro minimalizaci potenciálních problémů byla vytvořena testovací komponenta, jenž nepoužívá šablonu třídy pro práci s registry a omezuje se pouze na jeden registr a jednu akci. Registr prioritně

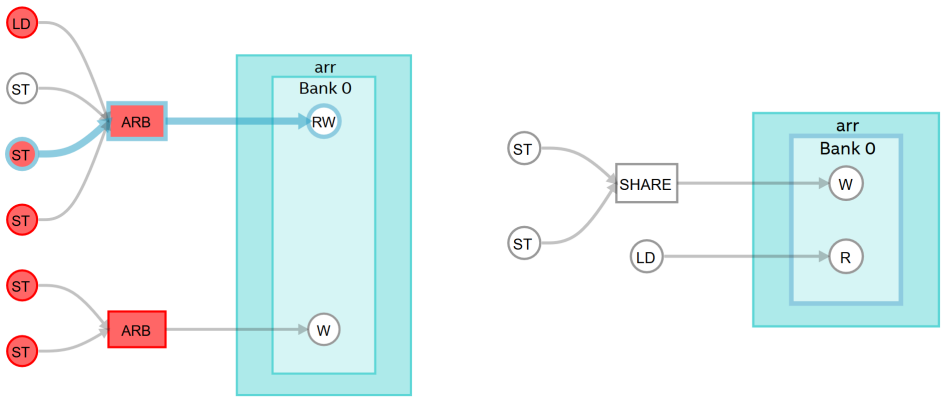
zpracovává čtecí či zápisové požadavky řídicí roviny, následně datové roviny. Datová rovina nejprve načítá data ze vstupu, tedy identifikátor akce, adresu a vstupní data. Následně je dle identifikátoru provedena patřičná akce, která násobí vstupní data a výsledek zapisuje do registru a na výstup datové roviny.

Registr je nastaven na šířku 40 bitů, hloubku 16 položek. Je nařízena implementace do blokové RAM, registr musí použít jednu banku o šířce 64 bitů a nesmí používat více než jeden čtecí a jeden zápisový port.

Očekávaná implementace registru by měla mít jednu banku o šířce 64 bitů a hloubce 16 položek. V rámci přístupu k registru by měly být dva sdílené přístupy do zápisového portu (řídicí a datová rovina) a jeden přístup do čtecího portu registru (řídicí rovina).

V případě reálné implementace registru byla vytvořena jedna banka hloubky 16 slov o šířce 64 bitů. Přístupů pro zápis bylo vytvořeno hned pět, pro čtení pak jeden. Přístupy jsou namapovány na jeden port pro čtení/zápis a jeden port pro zápis. O přístupu k portům rozhodují arbitry a celý přístup do paměti pozastavuje zbytek návrhu, čímž se zvyšuje II komponenty.

Kompilátor se v případě navrhované šířky 40 bitů snaží namapovat druhý přístup k zápisovému portu registru na tři přístupy o šířce 8 bitů a jeden přístup o šířce 16 bitů. Druhý přístup je tedy mapován na navrhovanou šířku registru 40 bitů, místo aby byl mapován na implementovanou šířku registru 64 bitů. Na obrázku 2.2 lze vidět špatnou implementaci přístupu do registru nalevo a správnou napravo.



Obr. 2.2: Případ špatné a správné implementace přístupu k registrům o šířce 40 bitů.

Při dalším testování byly ověřeny implementace registrů o šířce 41, 48, 49 a 64 bitů. V případě šířky 48 a 49 bitů došlo k špatné implementaci, zatímco u šířky 41 a 64 bitů došlo k správné implementaci. Výsledek implementace tedy závisí na šířce registru.

Problém přístupu řeší zarovnání nastavované hodnoty šířky registru na hodnotu implementované šířky, tedy je nutné zarovnávat šířku registrů na násobky 32. Problematika nesprávného přístupu k registrům byla oznámena výrobci kompilátoru.

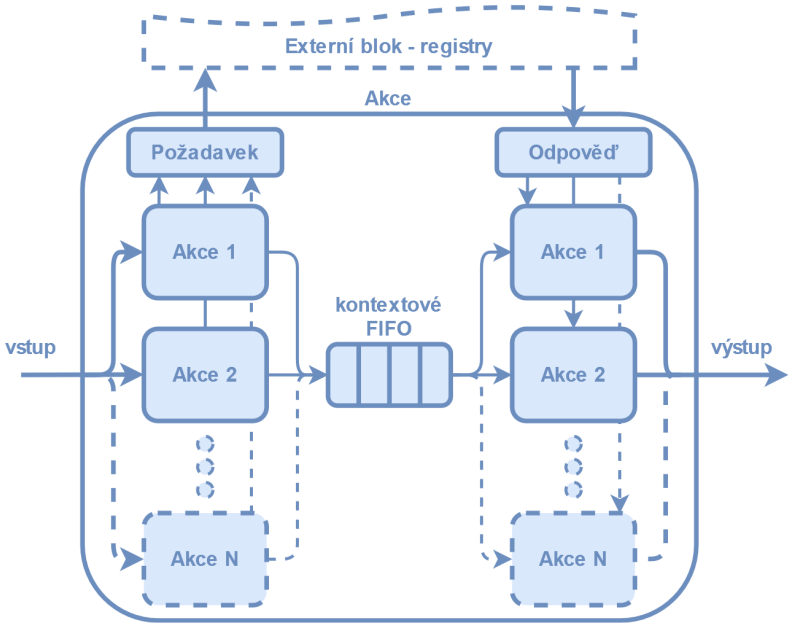
2.2.2 Akce, externí bloky a vzájemná komunikace

Následující část práce se věnuje návrhu akcí jenž provádí různé úkony a využívají externích bloků, jenž obsahují registry. Tato část práce se tedy zabývá návrhem akcí, externích bloků a jejich vzájemnou komunikací.

Akce

Akce (viz obrázek 2.3) v první části načítají vstup, tedy identifikátor akce, adresu a vstupní data. Pakliže je vstup načten, je provedena patřičná akce na základě jejího identifikátoru. Akce následně zapisuje požadavek na čtení či zápis do externího bloku a zároveň zapisuje identifikátor akce s adresou do kontextové FIFO paměti. Následně se druhá část akcí snaží načíst odpověď s daty z externího bloku. Pakliže je odpověď načtena, je načten i vstup kontextové FIFO paměti a na základě identifikátoru je provedena druhá část akce. V závěru jsou data zapsána na výstup.

Akce jsou rozděleny kontextovou FIFO pamětí, aby bylo možné nestále posílat požadavky externímu bloku a nebylo nutné čekat na jeho odpověď, která má určité zpoždění. Zpoždění navíc může být prodlouženo prioritním přístupem MI32 rozhraní k externím blokům.



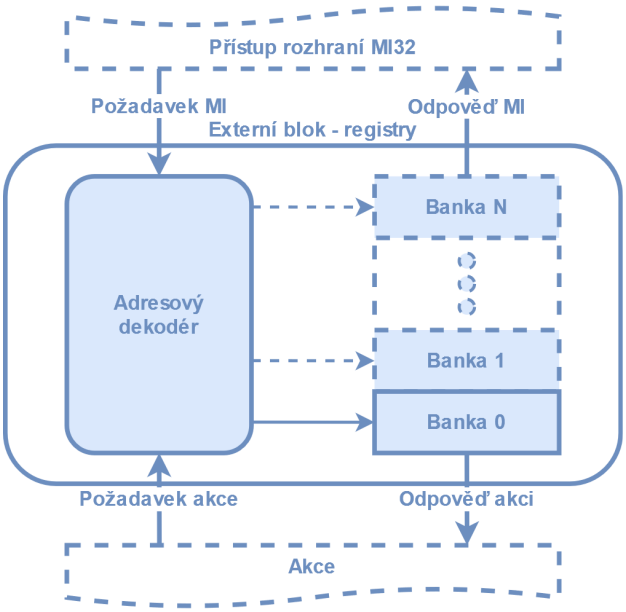
Obr. 2.3: Návrh akce.

Tento případ vystihuje návrh akcí, jenž pracují s jedním externím blokem. V případě práce s více externími bloky bude mít každá akce svou vlastní kontextovou FIFO paměť.

Externí blok

Externí blok (viz obrázek 2.4) se skládá z registrů a jejich přístupu, jenž je řešen skrze rozhraní pro přístup k více registrům. Prioritně jsou odbaveny požadavky řídicí roviny, tedy požadavky MI z přístupu MI32, a následně datové roviny, tedy požadavky akce.

Nejprve je načítán požadavek MI, tedy identifikátor operace, adresa a data. Pakliže je požadavek načten, určí se typ operace s registry, tedy zápis či čtení. Následně je na základě adresy vybrán cílový registr, kterým je daná operace provedena. Při čtení z registru je jeho hodnota zapsána do odpovědi MI. Pakliže není požadavek MI, je načítán požadavek akce. S požadavkem se poté pracuje jako v případě požadavku MI, pouze s tím rozdílem, že zápis do odpovědi akcím je proveden vždy.



Obr. 2.4: Návrh externího bloku.

Vzájemná komunikace

Do externího bloku přistupují dvě komponenty, tedy akce a přístup rozhraní MI32. Není však známo, v jakých okamžicích budou komponenty k bloku přistupovat, tím pádem obvod nelze staticky naplánovat a kompilátor by uvažoval nejhorší možnou

verzi. Komponenty jsou tedy od sebe odděleny rozhraním stream, které zajišťuje okolní komunikaci nezávislým čtením a zápisem každé komponenty.

Komponenty jsou následně spouštěny asynchronně, čehož je dosaženo díky systému úloh, který byl popsán v 1.4.2.

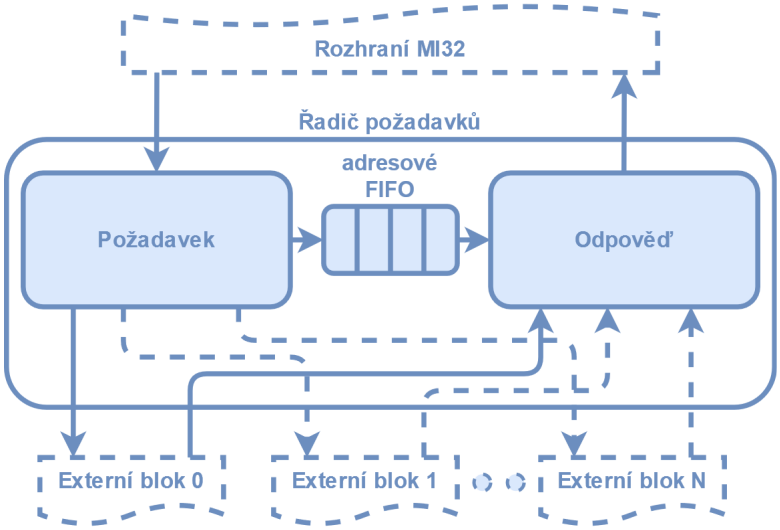
V rámci kódu jsou rozhraní *stream* pro komunikaci na globální úrovni. Externí blok a akce jsou implementovány jako funkce, které jsou volány komponentou *top* skrze systém úloh.

2.2.3 Přístup rozhraní MI32

Externí bloky je nutné propojit s rozhraním MI32, které zasílá požadavky fixní datové šířky 32 bitů. Nejprve je nutné směrovat požadavky do správného externího bloku, což dělá řadič požadavků. Následně musí mít externí blok převodník, který převádí datovou šířku 32 bitů na datovou šířku externího bloku a naopak.

Řadič požadavků

Řadič požadavků se stará o správné směrování požadavků z MI32 rozhraní do externích bloků a následné přeposlání odpovědi z externího bloku do MI32 rozhraní. Odpovědi na čtecí požadavky řadiče musí být odeslány ve stejném pořadí, jako byly odeslány požadavky, přičemž každý externí blok může mít rozdílnou latenci odpovědi, bez jejíhož uvážení by mohlo být narušeno správné pořadí odpovědí. Nejde tedy posílat čtecí požadavky do různých externích bloků okamžitě a řadič požadavků musí čekat na odpověď před zasláním dalšího čtecího požadavku do jiného externího bloku. Obrázek 2.5 zobrazuje návrh řadiče požadavků



Obr. 2.5: Návrh řadiče požadavků.

Řadič požadavků se skládá z dvou částí, které jsou propojeny interní FIFO pamětí s adresou čtecího požadavku. Funkcionalita jednotlivý částí je následující:

- **Část pro zpracování požadavku** nejprve načítá vstup MI32 rozhraní, který je složen z identifikátoru operace, adresy a dat. Pakliže je vstup načten, určí se pomocí adresového dekodéru externího blok, do kterého má být požadavek směrován. Jestliže jde o operaci čtení, je proveden zápis do adresové FIFO paměti propojené s částí pro zpracování odpovědi. Část pro zpracování odpovědi je spuštěna. Jestliže je požadavek čtení směřován do jiného externího bloku, část pro zpracovaní požadavku je pozastavena, dokud není odbavena předchozí odpověď.
- **Část pro zpracování odpovědi** nejprve načítá adresu z adresové FIFO paměti, podle které se snaží načíst odpověď z daného externího bloku. Jakmile je odpověď přijata, tak je přeposlána do MI32 rozhraní, ruší se pozastavení části pro zpracování požadavku a zastavuje se část pro zpracování odpovědi, dokud není znovu spuštěna částí pro zpracování požadavku.

Výpis 2.1 ukazuje ideální metodu instance propojení s externími bloky jako pole rozhraní stream o velikosti definované hodnoty *EXT_N*, což by zaručilo jednoduchý generický návrh. Pole rozhraní však není podporováno, je tedy nutné definovat přístupy k externím blokům jako samostatná rozhraní stream. Generický návrh bude následně zajištěn pomocí skriptu.

Výpis 2.1: Příklad ideální a reálné definice více rozhraní stream.

```
1 // Ideal method
2 ihc::stream_out<t_ext_req>          mi32_req[EXT_N];
3 ihc::stream_in<ac_int<32, false>>   mi32_res[EXT_N];
4 // Supported method
5 ihc::stream_out<t_ext_req>          mi32_req0;
6 ihc::stream_in<ac_int<32, false>>   mi32_res0;
7 ihc::stream_out<t_ext_req>          mi32_req1;
8 ihc::stream_in<ac_int<32, false>>   mi32_res1;
```

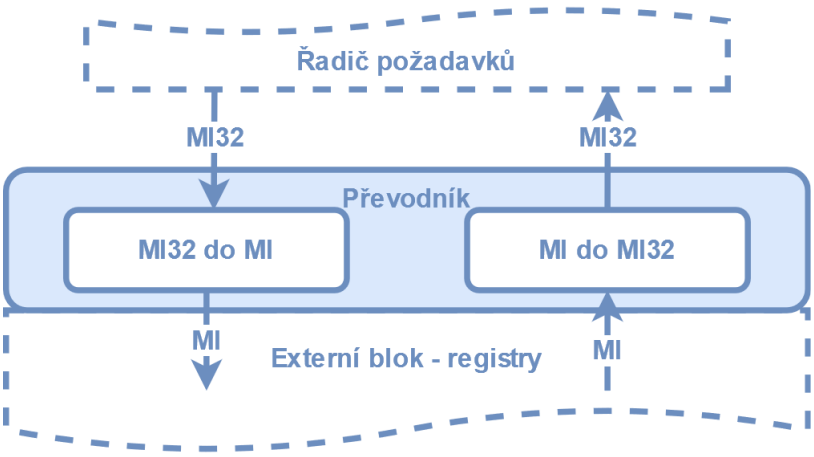
Převodník

Každý externí blok může mít různou datovou šířku, zatímco rozhraní MI32 má fixní datovou šířku 32 bitů. Je tedy nutné zajistit převod požadavků s datovou šířkou 32 bitů na větší datovou šířku a naopak, což dělá převodník.

Převodník (viz obr 2.6) je napojen na řadič požadavků, skrze který přicházejí požadavky z rozhraní MI32. Na druhé straně je řadič propojen s externím blokem,

do kterého zasílá požadavky převedené na datovou šířku daného externího bloku. Převodník je rozdělen do dvou samostatných komponent:

- **MI32 do MI** převádí požadavky MI32 datové šířky 32 bitů do požadavků MI datové šířky daného externího bloku.
- **MI do MI32** převádí odpovědi MI datové šířky externího bloku do odpovědi MI32 datové šířky 32 bitů.

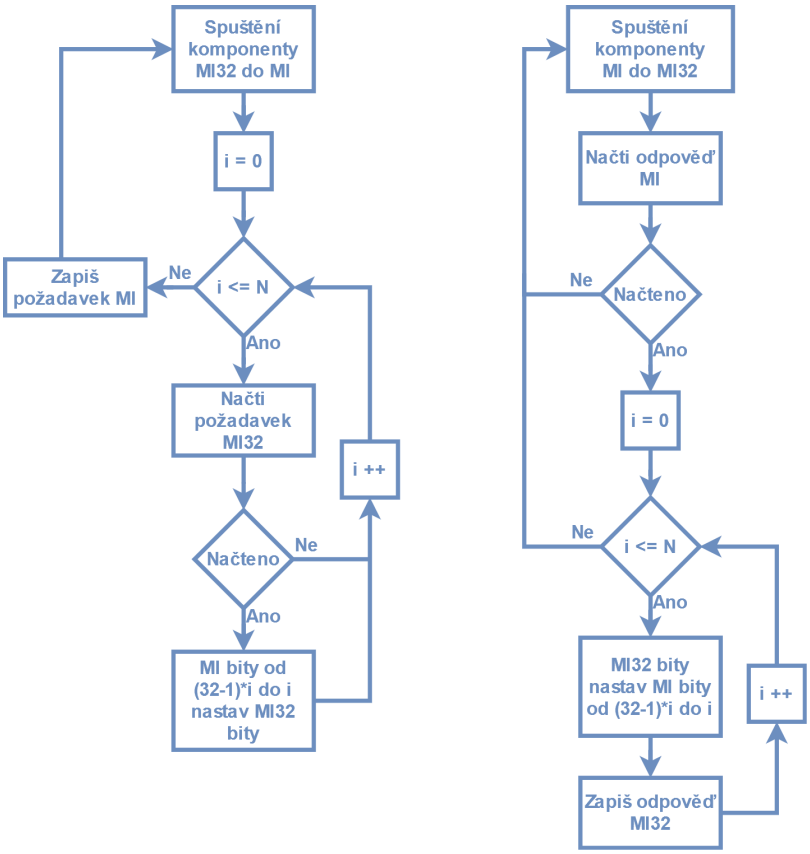


Obr. 2.6: Návrh rozhraní převodníku.

Pro otestování schopností kompilátoru bylo k návrhu převodníku přistupováno dvěma způsoby. Jeden způsob navrhuje převodník ve smyslu softwarového návrhu a využívá smyček. Druhý způsob navrhuje převodník ve smyslu hardwarového návrhu a je implementován jako konečný automat.

Převodník navržený v softwarovém smyslu lze vidět na obrázku 2.7 s blokovými diagramy komponent převodníku, kde na levé straně je převod požadavku MI32 na požadavek MI a na pravé straně je převod odpovědi MI do MI32. Funkcionalita komponent převodníku je následující:

- **Při převodu požadavku MI32 na MI** je ihned spuštěna smyčka, jenž načítá požadavek a v případě jeho načtení jsou MI bity od $(32-1)\times i$ do i nastaveny bity MI32. Jakmile je provedeno N cyklů smyčky, požadavek je zapsán na výstup MI a komponenta se spouští znovu. Parametr N určuje počet cyklů potřebných k vytvoření požadavku MI o datové šířce $32 \times N$ z požadavků MI32.
- **Při převodu odpovědi MI na MI32** je nejprve načtena odpověď MI a v případě jejího načtení je spuštěna smyčka, jenž MI32 bity nastavuje MI bity od $(32-1)\times i$ do i a odpověď se zapisuje na výstup MI32. Jakmile je provedeno N cyklů smyčky, komponenta se spouští znovu. Parametr N určuje počet cyklů potřebných k vypsání odpovědi MI32 z odpovědi MI o datové šířce $32 \times N$.



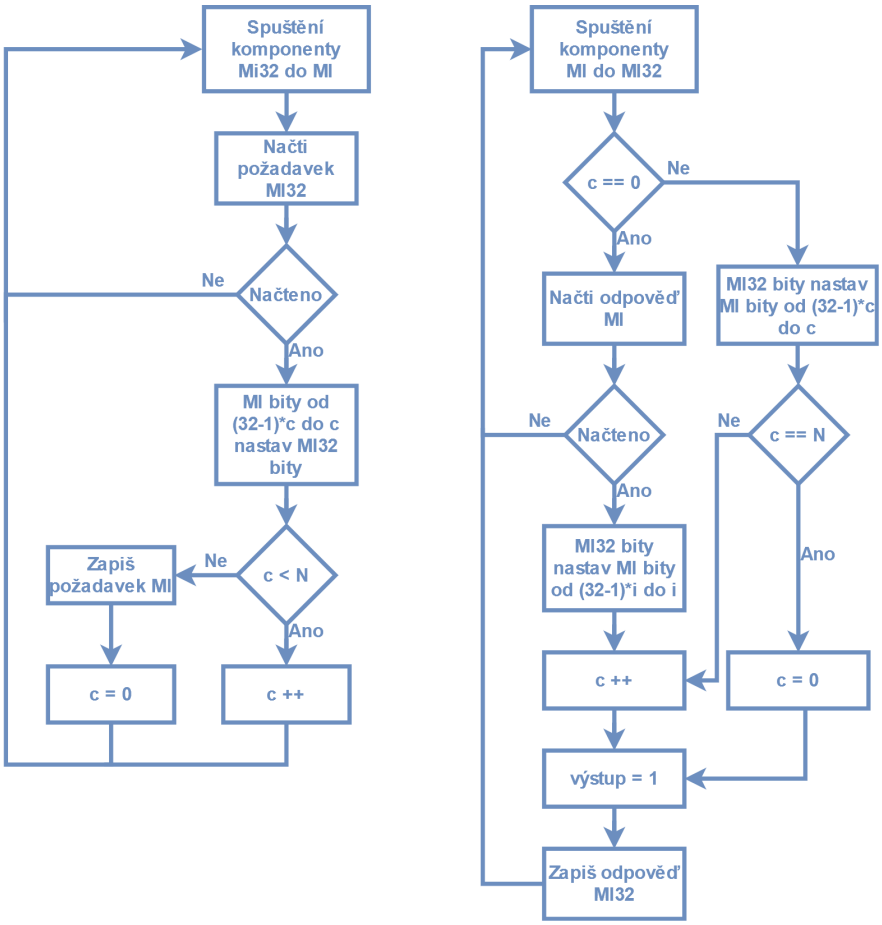
Obr. 2.7: Blokový diagram komponent převodníku ve smyslu softwarového návrhu.

Převodník navržený v hardwarovém smyslu lze vidět na obrázku 2.8 s blokovými diagramy komponent převodníku, kde na levé straně je převod požadavku MI32 na požadavek MI a na pravé straně je převod odpovědi MI do MI32.

Tento typ převodníku odstraňuje smyčky a místo toho využívá statického čítače (c) inicializovaného na hodnotu nula. Každá statická proměnná je implementována do registrů či pamětí, které jsou inicializovány pouze při prvním spuštění komponenty. Funkcionalita komponent převodníku je následující:

- **Při převodu požadavku MI32 na MI** je nejprve načten požadavek MI32 a v případě jeho načtení jsou MI bity od $(32-1) \times c$ do c nastaveny MI32 bity. MI bity jsou udržovány v registru. Následně je zkontrolováno, zdali má čítač hodnotu menší než parametr N, jenž určuje počet cyklů před zapsáním požadavku MI. Pakliže je podmínka splněna, je inkrementován čítač a komponenta zpracovává v dalším cyklu novou část požadavku MI32. V případě nesplnění podmínky je zapsán požadavek MI, čítač je vynulován a komponenta zpracovává v dalším taktu nový požadavek.
- **Pří převodu odpovědi MI na MI32** je nejprve zjištěno, zdali je odpověď MI načtena, tedy zdali je čítač roven nule. Pakliže ano, provede se načtení odpovědi

MI do registrů a v případě úspěšného načtení jsou MI32 bity nastaveny MI bity od $(32-1) \times c$ do c, je inkrementován čítač a je vystaven signál výstup, který umožní zápis odpovědi MI32. Komponenta následně zpracovává v dalším taktu novou část odpovědi. V případě kdy čítač není roven nule, probíhá práce s odpovědí MI uloženou v registrech stejným způsobem, dokud není čítač roven parametru N, jenž určuje počet cyklů potřebných k vypsání požadavku MI na požadavky MI32.



Obr. 2.8: Blokový diagram komponent převodníku ve smyslu hardwarového návrhu.

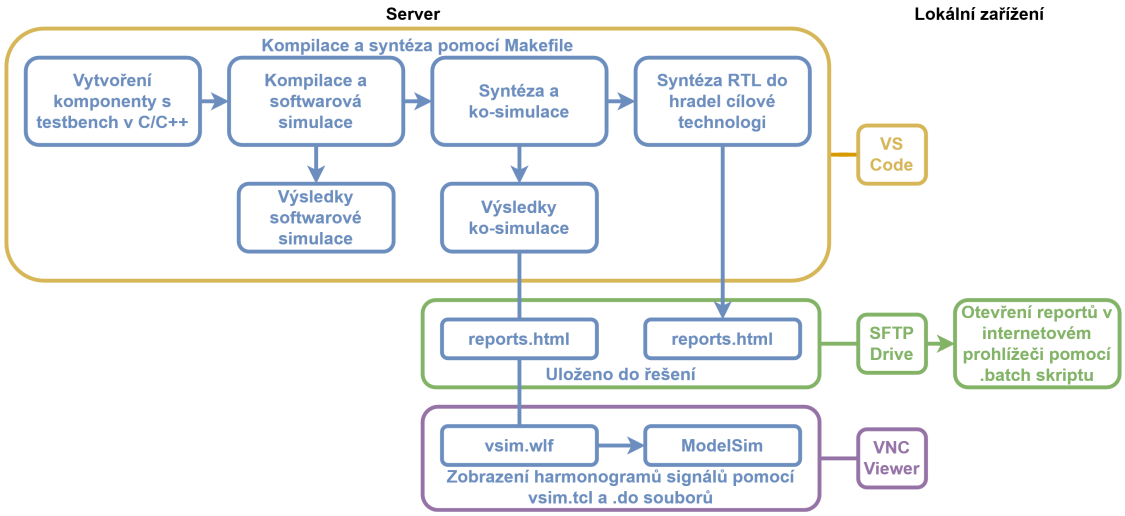
MI32 do MI v převodníku navrženém v softwarovém smyslu nefunguje dle očekávání, zatímco MI do MI32 ano. Problematika tkví v čtení požadavku uvnitř smyčky MI32 do MI, kterou kompilátor nedokáže při spuštění komponenty správně naplánovat. Převodník navržen v hardwarovém smyslu funguje a je implementován dle očekávání.

3 Vyhodnocení výsledků

V rámci vyhodnocení výsledků je nejprve popsáno testovací prostředí a vytvořené skripty pro usnadnění testování. Následně jsou popsány dosažené výsledky návrhů, které korespondují s navrženými komponentami (2.2) v rámci praktické části práce.

3.1 Testovací prostředí

Celý proces návrhu a kompilace probíhá na vzdáleném linuxovém serveru, kde je nainstalován kompilátor Intel HLS. K serveru přistupuje zařízení s operačním systémem Windows (dále jen lokální zařízení). Pro práci se serverem je používán program Visual Studio Code (dále jen VS Code), jenž přistupuje k serveru pomocí SSH (Secure Shell). VS Code má zabudovaný terminál, skrze nějž je možné pracovat se serverem. VS Code také obsahuje grafické rozhraní pro práci s adresáři a umožňuje editaci souborů vzdáleného serveru. V případě potřeby grafického uživatelského rozhraní (např. ModelSim) je na serveru spuštěn VNC Server, ke kterému je přistupováno programem VNC Viewer na lokálním zařízení. V případě práce s adresáři serveru na lokálním zařízení je použit program SFTP Drive. Testovací prostředí je lze vidět na obrázku 3.1.



Obr. 3.1: Testovací prostředí.

Kompilace Intel HLS se provádí zadáním specifických příkazů do terminálu. Aby příkazy nebyly pokaždé vypisovány, je vytvořen Makefile, který obsahuje volání těchto příkazů s přednastavenými parametry kompilace. Makefile také implementuje tzv. řešení (anglicky solutions), kdy za příkaz může být nastaven atribut s=N,

kde N je číslo souboru (*solution_N*), do kterého je uložen výsledek kompilace. Bez volby řešení je výsledek kompilace neustále přepisován do jednoho souboru, což v rámci porovnání výsledků různých architektur není žádoucí. Nad rámec práce poté byly vytvořeny skripty, které zjednodušují práci s těmito řešeními.

Reporty generované Intel HLS jsou uloženy do adresáře na vzdáleném serveru v podobě HTML (Hypertext Markup Language) souboru. Vzdálený server však nemá nainstalovaný prohlížeč. Přístup k reportům je tedy řešen skrze program SFTP Drive a následným otevřením reportu v prohlížeči lokálního zařízení. Pro jednoduchou práci je vytvořen *batch* skript, jenž automaticky otevírá reporty daných řešení. Skript má předdefinovanou kořenovou složku, jenž lze přepsat a následně je možné otevřít všechny reporty jednotlivých řešení současně nebo zvlášť. Skript preferuje výsledky syntézy pomocí Quartus Prime kvůli jejich větší informativnosti a přesnosti.

Intel HLS také generuje harmonogram signálů. Aby bylo signály možné zobrazit, je nutné přivolat ModelSim, po spuštění se navigovat do složky s harmonogramem signálů, přidat signály a z přidaných signálů zobrazit požadované signály. Pro ulehčení je vytvořen *tcl* skript, který implementuje práci s řešeními a obsahuje *do* soubory, které obsahují příkazy pro nastavení prostředí ModelSim. Nejprve se spustí ModelSim v adresáři navržené komponenty, kde je uložen skript v podobě *tcl* souboru. Tento skript se zavolá v příkazovém řádku ModelSim. Skript inicializuje prostředí prvním *do* souborem a následně dovoluje zobrazit harmonogramy signálů všech nebo jednoho řešení s využitím druhého *do* souboru.

3.2 Dosažené výsledky návrhů

Je nutné mít na paměti, že návrhy komponent spíše ověřují schopnosti kompilátoru, tedy jsou experimentální. Jinými slovy se návrhy zaměřují na správnou funkcionalitu a nehledí příliš na implementační detaily. Příkladem může být rozhraní pro přístup k více registrům, u kterého jde o správnou implementaci adresového dekodéru, avšak už nejde o počet registrů (implementační detail), ke kterým adresový dekodér přistupuje. K ověření funkcionality stačí registry dva. Dalším příkladem implementačního detailu může být počet propojení externích bloků u převaděče či výsledná šířka MI požadavku a odpovědi převodníku.

Výsledky navržených komponent jsou získány pomocí Intel HLS compiler Pro Edition, verze 19.4. Každý návrh je syntetizován do zdrojů cílového zařízení Arria 10 (součástka 10AX115U1F45I1SG, třída rychlosti -1). Délka hodinového signálu je nastavena na 10 nanosekund. V tabulkách jsou obsaženy následující parametry:

- II – inicializační interval udávající, po kolika taktech hodinového signálu může komponenta přijímat nová data.
- Latence – počet taktů hodinové signálu potřebných k vyřízení požadavku.

- Frekvence f [MHZ] – maximální frekvence v megahertz na které může komponenta pracovat.
- ALM (Adaptive Logic Module) – adaptivní logický modul.
- FF (Flip-Flop) – klopný obvod.
- RAM (Random Access Memory) – paměť s náhodným přístupem.
- DPS (Digital Signal Processing) unit – jednotka pro zpracování digitálního signálu.
- MLAB (Memory Logic Array Block) – paměť bloku logického pole.

Základní rozhraní pro přístup k registrům

Tabulka 3.1 zobrazuje výsledky syntézy základního rozhraní pro přístup k registrům s jedním registrem hloubky 8 slov o šířce 30 bitů.

Tab. 3.1: Výsledky syntézy základního rozhraní pro přístup k registrům.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
Top	1	5	311,33	335	278	0	0	0

Rozhraní pro přístup k více registrům

Tabulka 3.2 zobrazuje výsledky syntézy rozhraní pro přístup k více registrům. První registr je hloubky 8 slov o šířce 30 bitů, druhý je hloubky 16 slov o šířce 64 bitů. Intel HLS syntetizuje první registr do registrů, kvůli jeho malé velikosti. Druhý registr je syntetizován do paměti.

Tab. 3.2: Výsledky syntézy rozhraní pro přístup k více registrům.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
Top	1	12	520,02	459	979	2	1	1

Oproti základnímu rozhraní pro přístup k registrům lze vidět značné zvýšení latence, které je způsobeno adresovým dekodérem a dekodérem akce.

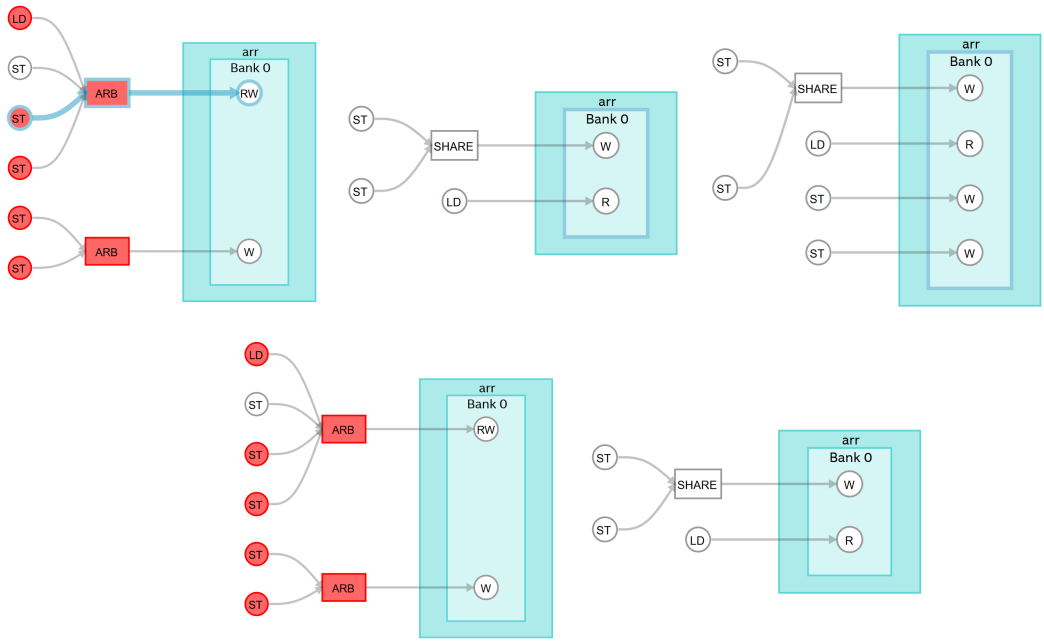
Problematika přístupu k registrům

Tabulka 3.3 zobrazuje výsledky syntézy modifikovaného základního rozhraní pro přístup registru, které bylo podrobně probráno v 2.2 praktické části práce. Je syntetizováno pět případů komponenty s rozdílnými šířkami registrů implementovaných do blokové RAM. Registry jsou hloubky 16 slov a šířky 40, 41, 48, 49 a 64 bitů.

Tab. 3.3: Výsledky syntézy modifikovaného základního rozhraní o různé šířce registru.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
Top (40b)	33	70	344,47	973,5	1446	4	0	22
Top (41b)	1	9	488,76	177,5	272	2	0	1
Top (48b)	3	12	322,58	202,5	654	3	0	1
Top (49b)	33	70	344,35	1053	1509	5	0	26
Top (64b)	1	9	467,07	180	316	2	0	1

Na obrázku 3.2 lze vidět jak jsou přístupy k registrům naplánovány. Intel HLS správně naplánoval přístup k registrům šířky 41 a 64 bitů. Chybně byly naplánovány přístupy k registrům šířky 40, 48 a 49 bitů, v jejichž případě značně narůstají výsledky všech parametrů návrhu, jen maximální frekvence klesá.



Obr. 3.2: Implementace přístupu k registrům o datové šířce 40, 41, 48, 49 a 64 bitů.

Akce propojené s externím blokem

Tabulka 3.4 zobrazuje výsledky syntézy akcí a externího bloku. Komponenty jsou volány obálkou top skrze systém úloh. Registry mají připravený adresový dekodér a obsahují jeden registr hloubky 8 slov o šířce 30 bitů, jenž je implementován do

registrů. Akce jsou rozděleny do dvou částí propojených kontextovou FIFO pamětí o hloubce 10 slov. V rámci každé části akce je dekodér akce. Jedna akce je zápisová a druhá čtecí.

Tab. 3.4: Výsledky syntézy akcí a externího bloku.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
Top	1	10	520,02	419	693	1	0	8

Řadič požadavků

Tabulka 3.5 zobrazuje výsledky syntézy řadiče požadavků, který je propojen s dvěma externími objekty.

Tab. 3.5: Výsledky syntézy řadiče požadavků.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
Top	1	5	507,36	136	143	0	0	0

Převodník

Tabulka 3.6 zobrazuje výsledky syntézy převodníku s komponentami MI32 do MI a MI do MI32. Zkratka SW označuje komponenty převodníku navrženého v softwarovém smyslu, HW poté v hardwarovém smyslu. Převodník je nastaven na převod požadavku/odpovědi MI32 fixní datové šířky 32 bitů na požadavek/odpověď MI datové šířky 64 bitů.

Tab. 3.6: Výsledky syntézy převodníku.

Komponenta	II	Latence	f [MHz]	ALM	FF	RAM	DSP	MLAB
MI32 do MI (SW)	1	7	645.16	160,5	260	0	0	1
MI do MI32 (SW)	2	7	645.16	85	135	0	0	0
MI32 do MI (HW)	1	5	645.16	56,5	105	0	0	0
MI do MI32 (HW)	1	5	645.16	41,5	69	0	0	0

Návrh části MI do MI32 v softwarovém smysl nefunguje dle očekávání, jelikož je při spuštění komponenty zapsán náhodný signál na výstup. Po prvotním výpisu však

komponenta pracuje jak má, výsledky tedy nebudou příliš ovlivněny špatným naplánováním a mohou alespoň poukazovat na dva přístupy návrhu hardware pomocí vysokoúrovňové syntézy.

Jak lze vidět, latence převodníku navrženého v hardwarovém a softwarovém smyslu se liší. Převodník navržen v softwarovém smyslu totiž není spuštěn znovu, dokud nejsou načtena či vypsána všechna data. V případě MI do MI32 je to potvrzeno inicializačním intervalem rovným dvěma, tedy komponenta dvakrát vypisuje, než je spuštěna znovu.

Oproti tomu je převodník navržen v hadrwarovém smyslu neustále spouštěn a data načítá či vypisuje pouze v moment, kdy je čítač nastaven do nuly. V případě MI do MI32 je to potvrzeno inicializačním intervalem rovným jedné, přestože komponenta načítá data každé dva takty. Díky tomuto přístupu je i značně snížena spotřeba zdrojů.

Závěr

Bakalářská práce je zaměřena na návrh komponent akcí a externích bloků v jazyce C/C++, které jsou následně syntetizovány nástroji pro vysokoúrovňovou syntézu (Intel HLS a Vivado HLS). Komponenty jsou navrženy tak, aby měly co nejmenší inicializační interval a mohly být jednoduše využívány P4 kompilátorem.

Teoretický úvod v první řadě seznámil s principy vysokoúrovňové syntézy, které jsou důležité pro správné uchopení práce s nástroji. Byl popsán jazyk P4, který se používá pro popis síťových prvků a umožňuje definice vlastních protokolů. Následně byl popsán kompilátor P4-to-VHDL jenž převádí program v jazyce P4 do jazyku VHDL. Bylo popsáno paměťové rozhraní MI32, jenž propojuje software s hardware, a je využito v rámci návrhu akcí a externích bloků. V poslední části teoretického úvodu byly popsány specifika kompilátoru Intel HLS, jeho způsob návrhu komponent, specifické konstrukce a kompilace.

Praktická část popisovala nejprve koncept návrhu akcí, externích bloků a přístupu MI32. Následně byly popsány navržené komponenty spadající do tří hlavních problematik. První problematika práce s registry vyřešila přístup k registrům, které zpracovávají požadavky jak z kontrolní roviny, tak datové. Práce s registry též popisuje a řeší problém implementace přístupu do paměti, který pochází ze špatné optimalizace paměti kompilátorem Intel HLS. Druhá problematika popisuje návrh akcí, externích bloků a vzájemné komunikace. Třetí problematika popsala návrh přístupu rozhraní MI32, jenž je vyřešen řadičem požadavků a převodníkem.

Ve vyhodnocení výsledků bylo popsáno testovací prostředí pro Intel HLS a následně byly popsány výsledky syntézy navržených komponent do zdrojů cílové technologie zařízení Arria 10.

V rámci práce se podařilo navrhnout koncept akcí, externích bloků a přístupu rozhraní MI32. Následně byly komponenty (akce, externí blok s registry, převodník a řadič požadavků) popsány pomocí jazyka C/C++ a syntetizovány kompilátorem Intel HLS. Komponenty syntetizované pomocí Intel HLS dosahují očekávaných výsledků a nemají inicializační interval větší, než je pro danou komponentu třeba. V rámci práce se nepodařilo otestovat navržené komponenty pomocí Vivada HLS, chybí tedy porovnání výsledků kompilátoru Intel HLS a Vivado HLS.

Komponenty jsou připraveny a v rámci navazující práce budou propojeny do jednoho funkčního celku. Následně bude celý systém syntetizován jak pro Intel HLS, tak pro Vivado HLS. Komponenty následně budou využity při tvorbě kompilátoru z jazyka P4 do C++/VHDL.

Literatura

[1] COUSSY, P., D.D. GAJSKI, M. MEREDITH a A. TAKACH. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers* [online]. 2009, **26**(4), 8-17 [cit. 2020-06-05]. DOI: 10.1109/MDT.2009.69. ISSN 0740-7475. Dostupné z: http://ieeexplore.ieee.org/document/5209958/

[2] REN, Haoxing. A brief introduction on contemporary High-Level Synthesis. In: *2014 IEEE International Conference on IC Design & Technology* [online]. Austin, TX: IEEE, 2014, 2014, s. 1-4 [cit. 2020-06-05]. DOI: 10.1109/ICICDT.2014.6838614. ISBN 978-1-4799-2153-9. ISSN 2381-3555. Dostupné z: http://ieeexplore.ieee.org/document/6838614/

[3] MARTÍNEK, Tomáš. *Pokročilé metody syntézy číslicových obvodů (High Level Synthesis)* [online]. 2018 [cit. 2020-06-05]. Dostupné z: http://www.fit.vutbr.cz/~korenek/grants.php.cs?file=%2Fproj%2F442%2FPublic%2FPrezentace%2FPU_53-2011-pcs-high_level_synthesis.pdf&id=442

[4] LAM, Y. M., J. G. F. COUTINHO, W. LUK a P. H. W. LEONG. Unrolling-based loop mapping and scheduling. In: *2008 International Conference on Field-Programmable Technology* [online]. Taipei: IEEE, 2008, 2008, s. 321-324 [cit. 2020-06-05]. DOI: 10.1109/FPT.2008.4762408. ISBN 978-1-4244-2795-6. Dostupné z: http://ieeexplore.ieee.org/document/4762408/

[5] Xilinx, Inc: *Vivado Design Suite User Guide, High-Level Synthesis* [online]. San Jose, May 30 2014 [cit. 2020-06-05]. Dostupné z: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[6] SINHA, Sharad a Thambipillai SRIKANTHAN. Dataflow graph partitioning for high level synthesis. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)* [online]. Oslo: IEEE, 2012, 2012, s. 503-506 [cit. 2020-06-05]. DOI: 10.1109/FPL.2012.6339265. ISBN 978-1-4673-2256-0. Dostupné z: http://ieeexplore.ieee.org/document/6339265/

[7] The P4 Language Consortium: *P4 16 Language Specification, version 1.1.0* [online]. November 30, 2018 [cit. 2020-06-05]. Dostupné z: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf

[8] BOSSHART, Pat, Dan DALY, Glen GIBB, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* [online]. 2014, **44**(3), 87-95 [cit. 2020-06-05]. DOI:

10.1145/2656877.2656890. ISSN 0146-4833. Dostupné z: https://dl.acm.org/doi/10.1145/2656877.2656890

[9] BENÁČEK, Pavel. *Generation of High-Speed Network Device from High-Level Description*. Praha, 2016. Disertační práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Hana Kubátová.

[10] The P4 Language Consortium: *The P4 Language Specification, Version 1.0.5* [online]. November 26, 2018 [cit. 2020-06-05]. Dostupné z: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

[11] BENÁČEK, Pavel, Viktor PUŠ a Hana KUBÁTOVÁ. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* [online]. Washington, DC: IEEE, 2016, 2016, s. 148-155 [cit. 2020-06-05]. DOI: 10.1109/FCCM.2016.46. ISBN 978-1-5090-2356-1. Dostupné z: http://ieeexplore.ieee.org/document/7544769/

[12] Testování. *Homeproj: Redmine for CESNET* [online]. Brno, 2018 [cit. 2019-01-03]. Interní webová stránka.

[13] MATOUŠEK, Jiří. *Implementace a verifikace vstupních a výstupních síťových bloků* [online]. Brno, 2009 [cit. 2020-06-08]. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav počítačových systémů. Vedoucí práce Jiří Tobola. Dostupné z: http://hdl.handle.net/11012/54541.

[14] SANTIAGO DA SILVA, Jeferson, François-Raymond BOYER a J.M. Pierre LANGLOIS. P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* [online]. New York, NY, USA: ACM, 2018, 2018-02-15, s. 147-152 [cit. 2020-06-05]. DOI: 10.1145/3174243.3174270. ISBN 9781450356145. Dostupné z: https://dl.acm.org/doi/10.1145/3174243.3174270

[15] Intel Corporation: *Intel® High Level Synthesis Compiler Pro Edition, User Guide* [online]. December 16, 2019 [cit. 2020-06-05]. Dostupné z: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/archives/ug-hls-19-4.pdf

[16] Intel Corporation: *Intel® High Level Synthesis Compiler Pro Edition, Reference Manual* [online]. February 10, 2020 [cit. 2020-06-05]. Dostupné

z: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/archives/mnl-hls-reference-19-4.pdf

[17] Mentor Graphics: *Algorithmic C (AC) Datatypes, software version v3.9.2* [online]. August, 2019 [cit. 2020-06-05]. Dostupné z: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf

Seznam symbolů, veličin a zkratek

SmartNIC	Smart Network Interface Controller
FPGA	Field Programmable Gate Array
RTL	Register-Transfer Level
HLS	High Level Synthesis
P4	Programming Protocol-Independent Packet Processors
VHDL	VHSIC Hardware Description Language
DFG	Data Flow Graph
CFG	Control Flow Graph
CDFG	Control Data Flow Graph
HDL	Hardware Description Language
RAM	Random Access Memory
ROM	Read-Only Memory
FIFO	First In First Out
ASAP	As Soon As Possible
ALAP	As Late As Possible
II	Initialization Interval
CPU	Central Processing Unit
ASIC	Application Specific Integrated Circuit
LPM	Longest Prefix Match
CESNET	Czech Educational and Scientific Network
PCAP	Packet Capture
TCAM	Longest Prefix Match
TCAM	Ternary Content-Addressable Memory
MI	Memory Interface
IP	Intellectual Property
QoR	Quality of Results
LSU	Load-Store Unit
API	Application Program Interface
SSH	Secure Shell
HTML	Hypertext Markup Language
ALUT	Adaptive Look-Up Table
FF	Flip-Flop
ALM	Adaptive Logic Module
DPS	Digital Signal Processing
MLAB	Memory Logic Array Block

A Obsah přiloženého CD

Přiložené CD obsahuje návrhy komponenty vytvořených v C++, skript pro spuštění kompilace, skripty pro lepší verifikaci a obrázky použité v rámci této práce.

- `/` ..... kořenový adresář přiloženého CD
  - `action_registers_separate` ..... akce, externí blok a vzájemná komunikace
    - `main.cpp`
    - `result.golden.dat`
  - `include` ..... hlavičkové soubory
    - `intel_registers.h`
    - `platform.h`
  - `memory_implementation` ..... problematika příštupu k registrům
    - `main.cpp`
    - `result.golden.dat`
  - `mi_converter_hw` ..... převodník navržený v hardwarovém smyslu
    - `main.cpp`
    - `result.golden.dat`
  - `mi_converter_sw` ..... převodník navržený v softwarovém smyslu
    - `main.cpp`
    - `result.golden.dat`
  - `obrazky` ..... obrázky použité v rámci práce
    - `abstraktni-model-preposilani.png`
    - `akce.png`
    - `algoritmy-planovani.png`
    - `deparser.png`
    - `extern.png`
    - `flattening.png`
    - `implementace-pameti-vse.png`
    - `koncept.png`
    - `mem-spravna-spatna.png`
    - `mi32-zapis-cteni.png`
    - `parser.png`
    - `planovani-smycek.png`
    - `prevodnik-hw.png`
    - `prevodnik-rozhrani.png`
    - `prevodnik-sw.png`
    - `radic.png`
    - `realne-planovani.png`
    - `ruzne-cdfg.png`
    - `struktura-pameti.png`
    - `testovaci-prostredi.png`
    - `typicka-architektura.png`
    - `zakladni-rozhrani.png`
  - `reg_action_with_mi` ..... základní rozhraní pro přístup k registru
    - `main.cpp`
    - `result.golden.dat`

- reg_multiple_act_with_mi ..... rozhraní pro přístup k více registrům
  - main.cpp
  - result.golden.dat
- scripts ..... skripty pro kompilaci a verifikaci
  - Makefile.mk. ... skript pro spuštění kompilace a syntézy s uložením do řešení
  - open_reports.bat ..... skript pro prohlížení reportů
  - vsim.tcl. ..... skript pro prohlížení harmonogramů signálů v ModelSim
  - vsim_init.do. ..... soubor s inicializačními příkazy pro ModelSim
  - vsim_sol.do. ..... soubor s příkazy pro vložení signálů do ModelSim
- splitter ..... řadič požadavků
  - main.cpp
  - result.golden.dat
- bakalarska_prace_xpanak04.pdf
- README.txt ..... informace ke kompilaci
- struktura_adresare.txt