VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

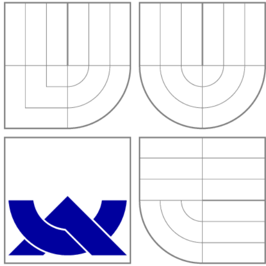
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MAPOVÁNÍ ALGORITMŮ DO TECHNOLOGIE FPGA S VYUŽITÍM NÁSTROJŮ VYSOKOÚROVŇOVÉ SYNTÉZY

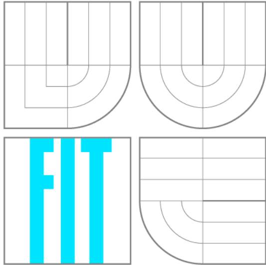
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE DAVID KUPKA
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MAPOVÁNÍ ALGORITMŮ DO TECHNOLOGIE FPGA S VYUŽITÍM NÁSTROJŮ VYSOKOÚROVŇOVÉ SYNTÉZY

MAPPING OF ALGORITHMS TO FPGA USING HIGH-LEVEL SYNTHESIS TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
DAVID KUPKA

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

BRNO 2011

Abstrakt

Tato práce se zabývá způsoby popisu hardware. Představuje metody používané při syntéze popisu a následně na sadě algoritmů porovnává dnes běžný nízkoúrovňový popis v jazyce VHDL s nově nastupující vysokoúrovňovou syntézou, kdy je komponenta popisována na algoritmické úrovni ve vyšším programovacím jazyce. Předmětem srovnání je poměr času potřebného pro implementaci a optimálnosti výsledné komponenty.

Abstract

This thesis deals with ways to describe hardware. It presents the methods used in the synthesis of the description and then it compares on a set of algorithms currently common low level description in VHDL with the newly emerging high-level synthesis, where a component is described at a algorithmic level in higher programming language. The object of comparison is the ratio of time required for implementation and optimality of the resulting components.

Klíčová slova

Vyskoúrovňová syntéza, VHDL, syntéza, popis hardware, srovnání

Keywords

High-Level Synthesis, VHDL, synthesis, hardware description, comparsion

Citace

David Kupka: Mapování algoritmů do technologie FPGA s využitím nástrojů vysokoúrovňové syntézy, bakalářská práce, Brno, FIT VUT v Brně, 2011

Mapování algoritmů do technologie FPGA s využitím nástrojů vysokoúrovňové syntézy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kořenka.

....
David Kupka
18. května 2011

Poděkování

Děkuji panu Ing. Janu Kořenkovi za cenné rady a trpělivé vedení při psaní této práce.

© David Kupka, 2011.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Techniky syntézy **5**
- 2.1 Reprezentace obvodu . . . 5
- 2.2 Základní transformace . . . 6
- 2.3 Plánovací algoritmy . . . 8
  - 2.3.1 ASAP . . . 9
  - 2.3.2 ALAP . . . 9
  - 2.3.3 Mobilita operátorů . . . 9
  - 2.3.4 Časově omezené plánování . . . 10
  - 2.3.5 Prostorově omezené plánování . . . 11
  - 2.3.6 Plánování smyček . . . 12
  - 2.3.7 Alokace zdrojů . . . 12

3 Hardwarová realizace algoritmů **14**
- 3.1 Prahový filtr . . . 14
  - 3.1.1 VHDL . . . 15
  - 3.1.2 Vysokoúrovňový nástroj . . . 16
- 3.2 Mediánový filtr . . . 17
  - 3.2.1 VHDL . . . 18
  - 3.2.2 Vysokoúrovňový nástroj . . . 18
- 3.3 Klasifikační jednotka . . . 18
  - 3.3.1 VHDL . . . 19
  - 3.3.2 Vysokoúrovňový nástroj . . . 20

4 Dosažené výsledky **21**
- 4.1 Prahový filtr . . . 21
- 4.2 Mediánový filtr . . . 22
- 4.3 Klasifikační jednotka . . . 22

5 Závěr **24**

A Obsah CD **26**

B RTL schémata **27**

Seznam obrázků

2.1 Nahrazení konstantních operací 6
2.2 Odstranění redudantních operátorů 7
2.3 Redukce výšky grafu 8
2.4 Sdružování operátorů 8
2.5 Force-Directed Heuristic 11

3.1 Schéma prahového filtru 15
3.2 Realizace prahového filtru ve VHDL 16
3.3 Schéma mediánového filtru 18
3.4 Schéma klasifikační jednotky 20

B.1 Schéma prahového filtru získaného syntézou VHDL popisu 28
B.2 Schéma prahového filtru získaného vysokoúrovňovou syntézou 29
B.3 Schéma mediánového filtru získaného syntézou VHDL popisu 30
B.4 Schéma mediánového filtru získaného vysokoúrovňovou syntézou 31

Kapitola 1

Úvod

V posledních desítkách let zasáhl rozvoj počítačů téměř do všech odvětví lidské činnosti. Velká část populace využívá ke své práci osobní počítače či notebooky a vyjímkou nejsou ani lidé, kteří vlastní více počítačů. Tyto počítače, které denně vidíme a používáme, jsou však pouze špičkou ledovce. Pokud se dobře rozhlédneme, najdeme počítač téměř v každé místnosti běžné domácnosti. V každém spotřebiči je vestavěn více či méně výkonný počítač, který řídí jeho činnost a zvyšuje uživatelský komfort.

Dle Moorova zákova se množství tranzistorů na čipu přibližně zdvojnásobí každých 18 měsíců, to znamená exponenciální růst složitosti čipů. Zároveň s komplexností stoupá spotřeba a (poněkud pomaleji) i výpočetní výkon procesorů. Na první pohled by se mohlo zdát, že za dobu, po kterou se procesory vyvíjejí, stoupl jejich výkon natolik, že lze řešit jakoukoli úlohu ve velice krátkém čase. Bohužel tomu tak není. Čím větší jsou možnosti počítačů, tím větší jsou na ně kladeny nároky. Navíc univerzálnost běžných procesorů omezuje jejich použití pro některé typy úloh. Zde vzniká prostor pro aplikačně-specifické čipy. Jako příklad nám poslouží kryptografie. V současné době přesahuje často hodnota dat hodnotu zařízení na kterém jsou uchována, je proto velmi vhodné tyto data patřičně chránit. Algoritmy používané v kryptografii (zejména pak asymetrické) jsou výpočetně velice náročné a šifrování na běžném procesoru může trvat velice dlouho. Pokud použijeme procesor speciálně navržený a optimalizovaný pro šifrování, zabere celá operace podstatně méně času a navíc, po dobu šifrování můžeme využívat plný výkon univerzálního procesoru. Obdobná situace dala vzniknout dedikovaným grafickým akcelerátorům.

Pokud zvážíme množsví tranzistorů, které společně vytvářejí čip, zjistíme, že není v lidských silách navrhnout rozmístění a vzájemné propoje všech tranzistorů tak, abychom získali požadované chování. Tento úkol bychom mohli přirovnat k požadavku vytvoření komplexního softwarového systému pouze s použitím strojového kódu. Proto s rostoucí složitostí navrhovaných komponent vznikají podpůrné nástroje s rostoucí mírou abstrakce. V současné době se pro popis hardwarových komponent používá skupina jazyků pro popis hardware (HDL, Hardware Description Language). V Americe dominuje jazyk Verilog, naproti tomu v Evropě je používanější jazyk VHDL. Tyto jazyky umožňují popisovat vytvářené komponenty na velmi nízké úrovni a dávají tak vývojáři plnou kontrolu nad vznikající architekturou. Nevýhodou tohoto přístupu je vysoká časová náročnost vytváření. Druhým možným přístupem je vysokoúrovňový popis, který umožňuje popisovat komponentu na úrovni algoritmu zapsaném v běžném programovacím jazyku. Implementační detaily jsou před vývojářem skryty, zajišťuje je nástroj pro vysokoúrovňovou syntézu (High-level Synthesis Tool). Tento přístup nabízí také zkrácení doby potřebné pro vývoj a nabízí tak vývojářům konkurenční výhodu v podobě včasného uvedení produktu na trh. Zvýšení abs-

trakce ale znamená odstínění implementačních detailů a automatické odvozování některých rozhodnutí, to může vést k neoptimálnímu výsledku.

Cílem této práce je zhodnotit poměr doby potřebné pro vývoj komponenty a její výsledné kvality prostředky vysokoúrovňové syntézy v porovnání s nízkoúrovňovým popisem pomoci jazyků pro popis hardware.

V první kapitole jsou popsány techniky používané při vysokoúrovňové syntéze, je představen způsob reprezentace popisu, popsány základní optimalizace, vysvětlen proces plánování a alokace zdrojů. Druhá kapitola představí vybrané algoritmy a jejich implementaci oběma přístupy. Ve třetí kapitole jsou srovnány a diskutovány dosažené výsledky.

Kapitola 2

Techniky syntézy

2.1 Reprezentace obvodu

Podobně jako u běžných programovacích jazyků je i u jazyků pro popis hardware velice nevýhodné, ne-li přímo nemožné, generovat přímo výsledný kód či architekturu. Vyvstává zde požadavek na formu reprezentace, nad kterou bude možné a nejlépe i nepříliš náročné provádět optimalizace. Jako vhodný prostředek se ukázal graf datových toků (Data-Flow Graph, DFG). Každý jednotlivý uzel DFG reprezentuje konkrétní operaci a hrany, spojující jednotlivé uzly, značí datové závislosti mezi operacemi.Na obrázku 2.1(a) vidíme graf datových toků, který značí datové závislosti v jednoduchém útržku VHDL kódu (2.1). Vidíme, že první a druhá operace mohou probíhat současně, protože na sobě nejsou datově závislé. Zatímco poslední operace může proběhnout až po splnění datových závislostí – spočítání obou operací sčítání.

Dále je potřeba reprezentovat vhodnou formou i řídící konstrukce. K tomuto účelu slouží graf řízení toku (Control-flow Graph, CFG). CFG reprezentuje rozhodovací logiku a vybírá, ve které z možných větví bude realizován výpočet. Každá z výpočetních větví je samostatný DFG. Jistou komplikací je zdržení výpočtu čekáním na vyhodnocení rozhodovací podmínky. Nabízí se možnost realizovat všechny větve výpočtu a zároveň vyhodnotit podmínku a až nakonec vybrat platný výsledek a ten propagovat dále. Tohoto přístupu můžeme v hardware oproti klasickým procesorům s výhodou využít. Toto však může být velice nevýhodné, pokud bychom chtěli sdílet zdroje dostupné na čipu mezi výpočetními bloky, protože tento přístup vyžaduje provádění všech bloků najednou.

Další možností je promítnout CFG do DFG. Výsledkem sloučení grafů je graf řízení a datových toků (Control Data Flow Graph, CDFG), dojde však ke sploštění grafu a není již přímo viditelné, která část grafu je řídící a která realizuje výpočet.

V praxi se používají různé modifikace CDFG, mezi které patří například:

Oddělené CDFG CFG a DFG jsou oddělené a CDF odkazuje na jednotlivé bloky. Výhodou je, že nedochází ke sploštění grafu, na druhou stranu je ale problematické provádět optimalizace na globální úrovni, protože každý funkční blok je samostatným DFG a jednotlivé bloky jsou propojeny pouze přes CFG.

DeJong's Graph Kontrolní informace jsou uloženy explicitně v DFG.

SSIM Graph CFG je kopií DFG a ukazuje pořadí provádění jednotlivých operací.

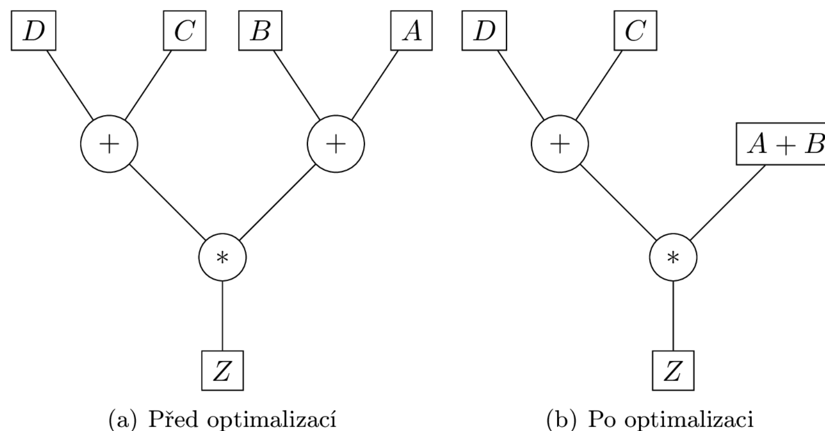
```
1  X <= A + B;
2  Y <= C + D;
3  Z <= X * Y;
```

Ukázka kódu 2.1: Příklad

2.2 Základní transformace

Nyní, když máme popis v programovacím jazyce reprezentován některým z výše popsaných grafů, můžeme, podobně jako při překladu kódu běžného programovacího jazyka pro procesor, provést řadu optimalizací. Každá optimalizace je transformací nad grafem. Mezi základní transformace patří:

Vyhodnocení výrazů s konstantami Pokud se v grafu vyskytuje operace, která má na všech svých vstupech konstanty, je výhodné předpočítat si výsledek této operace a celý výraz nahradit jedinou konstantou. Představme si, že proměnné A a B z příkladu 2.1 jsou konstanty. Bez optimalizací vznikne DFG, který můžeme vidět na obrázku 2.1(a). V tom případě by DFG po provedení této optimalizace vypadal podobně jako na obrázku 2.1(b).

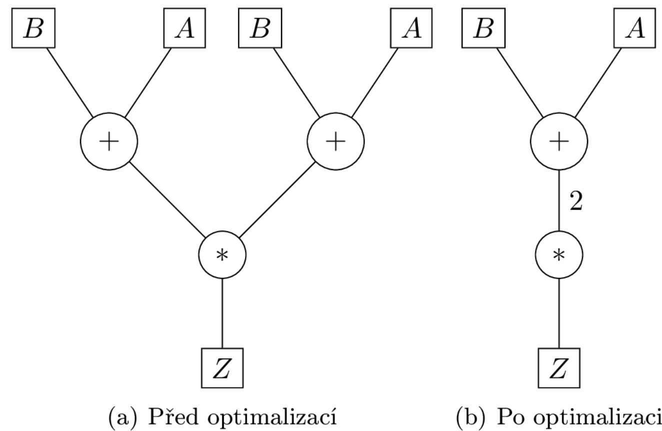


(a) Před optimalizací (b) Po optimalizaci

Obrázek 2.1: Nahrazení konstantních operací

Eliminace mrtvého kódu Jestliže je část grafu nedosažitelná za všech podmínek, bylo by nevýhodné tuto část překládat a následně syntetizovat. Nikdy nepoužitý blok by zbytečně zabíral zdroje a také by mohl negativně ovlivnit maximální dosažitelnou frekvenci.

Odstranění redudantních operátorů V grafu se také mohou vyskytovat vícekrát stejné operace s identickými vstupy. Takový výpočet je nadbytečný a pokud není pro funkci nezbytný (např. detekce poruchy rozdílným výsledkem stejného výpočtu), je vhodné provést operaci pouze jednou a její výsledek rozvést do všech míst, kde se původně redudantní operace vyskytovaly. Pokud by v příkladu 2.1 byly u obou operací shodné parametry, bylo by možné tuto opreci provést jen jednou. Neoptimalizovaný DFG vidíme na obrázku 2.2(a), optimalizovaný pak na obrázku 2.2(b). Na první pohled je zřejmá úspora jednoho sčítacího operátoru a části propojovací sítě.



(a) Před optimalizací (b) Po optimalizaci

Obrázek 2.2: Odstranění redudantních operátorů

Vytknutí společných částí před smyčku Pokud je možné část kódu, vyskytujícího se ve smyčce, vytknout před tuto smyčku, je vhodné tak učinit. Snížíme počet kontrolních kroků uvnitř smyčky a v případě rozbalení smyčky ušetříme i zdroje dostupné na čipu.

Přiřazení atributu signálu Některé konstrukce vyjadřují určitou vlastnost, kterou je možné vyjádřit složitým aritmeticko-logickým výrazem. To však většinou není potřeba a tato konstrukce je nahrazena přiřazením atributu signálu. Typickým zástupcem těchto atributů je náběžná hrana hodinového signálu. Náběžnou hranu hodinového signálu je možné detekovat jako dvojici současně platících podmínek. Jednou z nich je výskyt změny úrovně signálu (exkluzivní logický součet zpožděného signálu s nezpožděným) a druhou pak vysoká úroveň signálu.

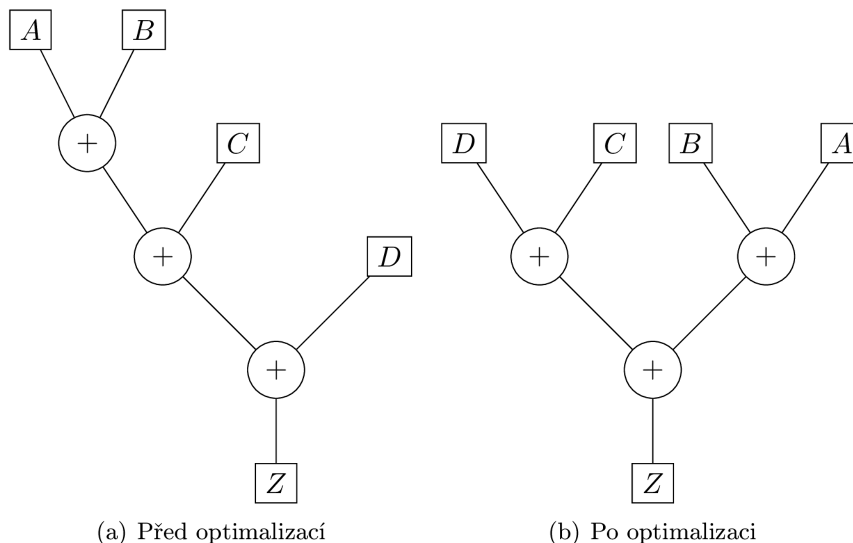
Redukce výšky grafu Využitím komutativních, distributivních a asociativních vlastností operátoru můžeme redukovat výšku stromu a tím zkrátit délku výpočtu s využitím paralelizmu. Pokud vytvoříme DFG z VHDL kódu v příkladu 2.2, vznikne nám graf, který můžeme vidět na obrázku 2.3(a). Využitím komutativnosti operace sčítání získáme DFG na obrázku 2.3(b). Problém může nastat v případě, že bychom tyto operace prováděli s čísly v plovoucí řádové čárce. I přes snahu vývojářů není komutativnost těchto operací zaručena.

```
1            X <= A~+ B;
2            Y <= V~+ C;
3            Z~<= W + D;
```

Ukázka kódu 2.2: Příklad

Transformace CFG do DFG Vyhodnocování podmínek z CFG lze promítnout do DFG. Dochází ke zkrácení doby výpočtu za cenu sploštění grafu a nemožnosti sdílet operátory mezi nezávislými zdroji. Všechny výpočetní větve i vyhodnocování podmínek CFG probíhá současně a výsledek je vybrán nakonec.

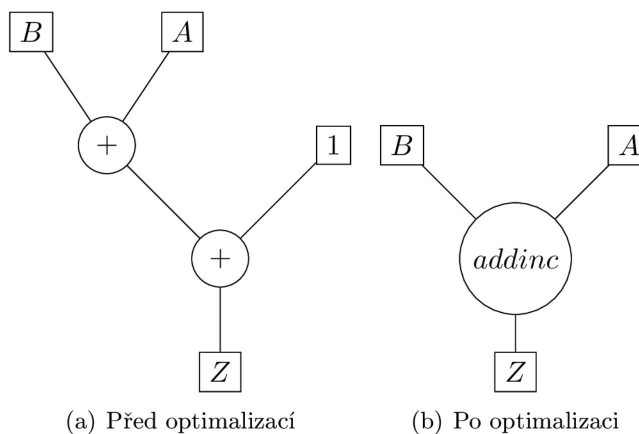
Sdružování operátorů V případě, že ve výpočtech využíváme často skupinu stejných operací, můžeme je združit do jedné a tím ušetřit zdroje i čas potřebný k vyhodnocení výrazů. Sdružovat můžeme obecně libovolné množství operací do jednoho uzlu, vzniká obecná



(a) Před optimalizací (b) Po optimalizaci

Obrázek 2.3: Redukce výšky grafu

aritmeticko-logická jednotka. Mějme situaci, kdy v našem algoritmu často provádíme operaci sčítání a následně inkrementaci. Tato operace bude vyžadovat dva kontrolní kroky. Jelikož máme plnou kontrolu nad hardwarovou architekturou, kterou vytváříme, můžeme si definovat vlastní operaci, která bude provádět obě tyto operace najednou v jednom kontrolním kroku. Situace je znázorněna na obrázku 2.4(a).



(a) Před optimalizací (b) Po optimalizaci

Obrázek 2.4: Sdružování operátorů

2.3 Plánovací algoritmy

Plánování je úloha rozdělující operace do jednotlivých kontrolních kroků. Na vstupu plánovacího algoritmu máme CDFG, ten je dělen do podgrafů, které jsou naplánovány do jednoho kontrolního kroku. V každém podgrafu tak může být nejvýše tolik operátorů každého typu, kolik jich je k dispozici v cílovém čipu. Různé podgrafy mohou sdílet funkční jednotky mezi kontrolními kroky. Proces plánování výrazně ovlivňuje množství použitých zdrojů i počet kontrolních kroků potřebných k realizaci celého grafu.

2.3.1 ASAP

ASAP (As Soon As Possible, "Co nejdříve") je velice jednoduchý algoritmus, který vybírá pro naplánování operátoru nejbližší možný kontrolní krok. V prvním kroku nalezne veškeré uzly bez datové závislosti na jiném uzlu, tyto naplánuje do prvního kontrolního kroku. Následně nalezne všechny uzly, jejichž datové závislosti jsou splněny naplánováním předka do některého z předchozích kontrolních kroků, a naplánuje je do kroku o jedna většího než je maximum jeho předků. Takto algoritmus pokračuje, dokud nejsou všechny uzly naplánovány. Jistou nevýhodou tohoto algoritmu je nevyváženost požadavků na zdroje. Typicky je plánováno mnohem více operátoru do prvních kontrolních kroků než do posledních.

foreach *uzel* $v_i \in V$ **do**
  if $Predci_{v_i} = \emptyset$ **then**
    $KontrolniKrok_i = 1;$
    $V = V - \{v_i\};$
  else
    $KontrolniKrok_i = 0;$
  end
end
while $V \neq \emptyset$ **do**
  foreach *uzel* $v_i \in V$ **do**
    if $vsichniPredci_{v_i} naplanovani$ **then**
      $KontrolniKrok_i = max(Predci_{v_i}, KontrolniKrok) + 1;$
      $V = V - \{v_i\};$
    end
  end
end

Algoritmus 1: ASAP

2.3.2 ALAP

ALAP (As Late As Possible, "Co nejpozději") je v podstatě opakem algoritmu ASAP. Motivací pro tento postup je snaha vyvážit situaci, kdy obecně při výpočtu redukujeme množství dat, a proto je množství operátorů použitých ke konci výpočtu nižší než na začátku. Postup algoritmu je tedy následující: V prvním kroku nalezne všechny uzly, které nemají žádného následníka a naplánuje je do kontrolního kroku T, poté nalezne všechny uzly jejichž následníci jsou již naplánování a tyto naplánuje do kontrolního kroku T-1, takto pokračuje, dokud nejsou všechny uzly naplánovány. Poslední kontrolní krok T-N, nám dává informaci o počtu potřebných kontrolních kroků jako N+1.

2.3.3 Mobilita operátorů

Oba výše uvedené algoritmy jsou spíše teoretické a v praxi se používají pouze jako pomocné výpočty pro složitější plánovací algoritmy. Jejich důležitou vlastností je, že určují horní (ASAP) a spodní (ALAP) hranici kontrolního kroku, do kterého může být konkrétní operace naplánována. Aritmetický rozdíl těchto hranic určuje mobilitu operátoru. Mobilita operátoru tedy vymezuje interval kontrolních kroků, do kterých mužeme operaci naplánovat.

```
foreach uzel v_i ∈ V do
    if Naslednici_{v_i} = ∅ then
        KontrolniKrok_i = T;
        V = V − {v_i};
    else
        KontrolniKrok_i = 0;
    end
end
while V ≠ ∅ do
    foreach uzel v_i ∈ V do
        if vsichniNaslednici_{v_i} naplanovani then
            KontrolniKrok_i = min(Naslednici_{v_i}, KontrolniKrok) − 1;
            V = V − {v_i};
        end
    end
end
```

Algoritmus 2: ALAP

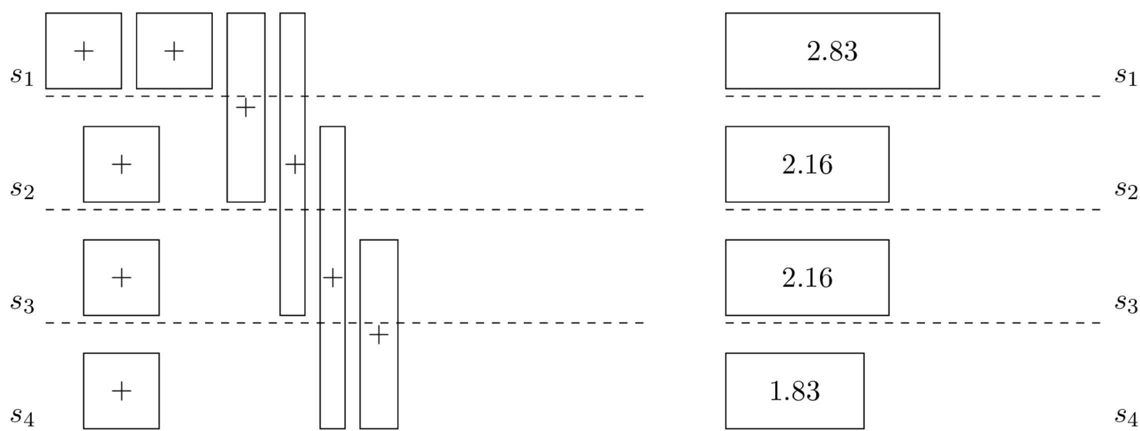
2.3.4 Časově omezené plánování

Pro některé aplikace je kritické striktní dodržení doby zpracování vstupních dat. Například DSP aplikace většinou pracují s určitou vzorkovací frekvenci a je velice žádoucí, aby zpracování aktuálního vzorku bylo dokončeno ještě před příchodem dalšího vzorku. Při známé frekvenci hodinového signálu získáme maximální počet kontrolních kroků, do kterých je nutné naplánovat věškeré uzly grafu. Toho bychom mohli často dosáhnout masivní paralelizací, avšak ekonomické důvody (nižší cena, menší energetická náročnost, vyšší efektivita) vytvářejí snahu o minimální spotřebu výpočetních zdrojů.

Přesné určení optimálního plánu je možné, avšak je třeba vyhodnotit veškeré kombinace, tudíž má tato metoda exponenciální časovou složitost a pro reálné systémy je prakticky nepoužitelná. Je tedy nutné použít heuristik, které nezaručují optimální řešení, ale podstatně zkracují dobu potřebnou pro plánování.

Force-Directed heurictic Jednou z používaných heuristik je Force-Directed heuristic. Cílem této metody je rovnoměrně rozdělit operace stejného typu mezi kontrolní kroky, díky tomu se zvýší využití jednotlivých funkčních bloků na čipu. Pro každý vrchol CDFG se na základě jeho mobility vypočte pravděpodobnost, že bude umístěn do daného kontrolního kroku. S rostoucí mobilitou klesá pravděpodobnost umístění do kroku. V případě umístění operátoru je jeho pravděpodobnost změněna na 1 v kroku, do kterého byl umístěn a na 0 v krocích, kde umístěn nebyl. Cílem heuristiky je, aby součty pravděpodobností jednotlivých kontrolních kroků byly rovny nebo alespoň jejich rozdíly byly co nejmenší. Po každém rozhodnutí o umístění je potřeba ověřit, zda se nezměnila mobilita některé z datově závislých operací.

Integer linear programming U tohoto plánovacího algoritmu jsou podmínky exaktně zadány soustavou lineárních nerovnic, které se řeší běžným matematickým aparátem. Tato metoda je vhodná pouze pro malé grafy, protože má vysokou časovou náročnost. Vede však na optimální řešení.



Obrázek 2.5: Force-Directed Heuristic

Iterative Refinement Při použití této metody plánování je některou jednodušší metodou vygenerován počáteční plán. V následujících krocích se každá operace postupně plánuje do všech přípustných kontrolních kroků a vybere se takový posun, který je nejvýhodnější. Tento stav je počátečním stavem pro další iteraci. Výpočet probíhá, dokud jsou získávány lepší výsledky.

2.3.5 Prostorově omezené plánování

Jiným způsobem plánování je prostorově omezené plánování, kdy máme typicky k dispozici nějaký konkrétní čip a můžeme využít všechny v něm dostupné zdroje. Našim cílem je pak co možná nejmenší počet kontrolních kroků. Obecný postup této skupiny algoritmů je vytvoření plánů s minimálním počtem zdrojů a následné přidávání dalších zdrojů tak, abychom redukovali počet kroků, ale nepřesáhli počet dostupných zdrojů.

List-Based Scheduling List-Based Scheduling vychází z algoritmu ASAP, je doplněn omezením počtu operátorů jednotlivých typů. Pro jednotlivé operátory se v každém kroku udržuje seznam vrcholů, které je možné naplánovat, tzn. uzly bez předků nebo s již naplánovanými předky. Tyto se seřadí podle priority a ze seznamu se vyberou vrcholy, které se naplánují na dostupné zdroje. Tím vzniknou další plánovatelné uzly, které se opět přidají do prioritních seznamů. Algoritmus pokračuje, dokud nejsou veškeré vrcholy naplánovány. Optimálnost této plánovací funkce závisí výrazně na použité prioritní funkci. Mezi možné vstupy prioritní funkce patří:

Mobilita Upřednostňují se uzly s menší mobilitou, protože mají omezené možnosti vzhledem k dalšímu plánování.

Délka datové cesty Vrcholy s delší cestou ke konci grafu jsou upřednostněny, aby celkový počet kontrolních kroků byl co nejnižší.

Množství následníků vrcholu Vrcholy s větším počtem následníků jsou plánovány dříve, získáváme více možností dalšího plánování.

2.3.6 Plánování smyček

Smyčky se v běžných algoritmech vyskytují velice často. Každá smyčka se skládá z úvodní části (prolog), kde probíhá inicializace proměnných, vlastního výpočetního jádra smyčky, které se provádí buďto předem známý počet iterací nebo do splnění zadaných podmínek, a závěrečné části (epilog), kde jsou předány výsledky výpočtu k dalšímu zpracování. Optimálnost naplánování smyčky je možné posuzovat podle dvou kritérií:

Využití dostupných zdrojů Pokud se rozhodneme na čip umístit nějaký funkční blok, je vhodné ho co nejvíce využít, jinak se dopouštíme plýtvání a naše architektura je neefektivní. Toto kritérium lze vyčíslit poměrem využitých funkčních bloků ku všem dostupným blokům.

Počet kontrolních kroků na iteraci Udává se jako poměr kontrolních kroků k počtu iterací smyčky.

Rozbalování smyček Jednou z možností plánování smyček je rozbalení smyček. Tento postup lze použít, pouze pokud předem známe počet iterací smyčky. Rozbalení smyčky provedeme duplikací DFG a následným plánováním běžným způsobem. Výsledný graf je složitější a plánování tudíž trvá delší dobu. Nevýhodou postupu je ztráta informace o jednotlivých iteracích. Jsou naplánovány přes sebe a není jednoduše rozlišitelné, která operace patří do které iterace.

Skládání smyček Skládání smyček využívá dělení smyčky na prolog, jádro a epilog. Metoda předpokládá, že prolog a epilog bude proveden jen jednou a výpočetní jádro bude provedeno mnohokrát, takže režie spojená s epilogem a prologem bude zanedbatelná. Jednotlivé části smyčky se překrývají podobně jako při zřetězení. Metodu lze použít i pro smyčky s předem neznámým počtem iterací.

Plánování vnořených smyček Vnořené smyčky jsou velice dobře paralelizovatelné. Pro plánování je nutné, aby smyčky byly tzv. dokonale vnořené. Dokonale vnořená smyčka je taková, která provádí výpočet pouze v nejvnitřnější smyčce. Obecné smyčky lze převést do tvaru dokonale vnořených, tento postup je reverzní k vytýkání operací před smyčku.

Výpočet uvnitř smyčky lze reprezentovat jako výsledek funkce v závislosti na hodnotách vypočtených v předchozích krocích. Pro proces plánování není vůbec důležitá funkce, která počíta hodnotu aktuální proměnné, ale spíše datové závislosti. Rozbalením všech iterací dostáváme iterační prostor. Iterační prostor je polytop[1] v n-rozměrném prostoru, kde počet rozměrů n je roven počtu vnořených smyček. Doplněním datových závislostí do iteračního prostoru získáváme graf datových závislostí. Tyto principy lze aplikovat pouze na statické smyčky. U dynamických smyček je i iterační prostor dynamický.

2.3.7 Alokace zdrojů

Závěrečnou úlohou je alokace zdrojů, jejímž úkolem je přiřadit konkretní operace konkrétním výpočetním elementům. Je velice vhodné volit funkční jednotky, paměťové elementy i prvky propojovací sítě vhodně s ohledem na výsledné časování obvodu. Vzdálenost datově závislých funkčních elementů lze přirovnat k lokalitě programu pro procesor. Pokud program obsahuje mnoho vzdálených skoků, stoupá režie spojená s prováděním programu

[1] Prostorový útvar omezený lineárními plochami

a klesá efektivita. Podobně, čím jsou datově závislé elementy vzdálenější od sebe, tím delší propoje vyžadují. Tím narůstá zpoždění, klesá frekvence a zvyšuje se cena. Alokace lze rozdělit do tří hlavních částí:

Přiřazení funkčních jednotek Obecně je k dispozici pro každou operaci více funkčních jednotek a úkolem alokace je vybrat nejvhodnější. Vhodnost jednotky ovlivňuje mnoho vlivů, například umístění na čipu, efektivita využití prvků atd.

Přiřazení paměťových elementů Úkolem je namapovat proměnné, konstanty a datové struktury do vhodných paměťových prvku. Konstanty je vhodné mapovat na ROM[2] prvky. Proměnné a struktury do RAM[3] pamětí či registrů. Na základě doby života proměnné[4] je možné sdílet paměťové pozice nebo registry mezi více proměnnými. Je také možné sdružovat registry do registrových polí.

Přiřazení propojovacích sítí Každý datový přenos je potřeba realizovat propojením zdrojového a cílového prvku, tyto propojovací cesty mohou být sdíleny více nezávislými přenosy, pokud neprobíhají současně. Cílem alokačního algoritmu je sdílet co možná nejvíce propojů a snížit tak cenu propojovací sítě.

Tato kapitola byla zpracována na základě informací z knih [4, 5] a záznamu přednášky [9].

[2]Pamět pouze pro čtení, Read-only memory

[3]Paměť s náhodným přístupem, Random access memory

[4]Interval mezi prvním a posledním použitím proměnné. Udává se v počtu kontrolních kroků.

Kapitola 3

Hardwarová realizace algoritmů

Obecně je možné implementovat jakýkoli problém v hardware, avšak některé algoritmy jsou vhodnější. Typicky je vhodné hardwarově akcelerovat algoritmy, které provádějí omezenou množinu operací nad velkým objemem dat. Při tvorbě komponenty můžeme její architekturu a rozhraní definovat podle potřeb dané aplikace. První skupinou algoritmů jsou algoritmy zpracovávající obraz, typicky totiž nad velkým množstvím obrazů provádíme jednu operaci (filtrování, detekce hran, atd.). Druhou skupinu tvoří síťové algoritmy, kde abychom plně využili rychlost připojené linky, musíme zpracovávat velké množství dat. Typickými síťovými algoritmy je vyhledání nejdelšího odpovídajícího prefixu pro směrovací rozhodování nebo srovnávání jednotlivých polí přenášeneného paketu s uloženými informacemi za účelem získání filtrovacího pravidla firewallu.

3.1 Prahový filtr

V mnoha metodách rozpoznávání obrazu je nutné odlišit body patřící objektu od bodů okolí. Pokud je úroveň jasu bodů objektu dostatečně odlišná od úrovně jasu okolí, je prahování i přes svou jednoduchost velice efektivní metodou segmentace obrazu [10]. Prahový filtr se nejčastěji používá pro převod šedotónového obrazu na černobílý. U barevného obrazu je prahování aplikováno na každý barevný kanál zvlášť nebo je k převodu použita intenzita bodu, která je sumou hodnot jednotlivých barevných kanálů vynásobených empiricky zjištěnými konstantami. Konkrétní hodnoty jsou uvedeny ve vzorci 3.1, kde I je výsledná intenzita bodu, R, G a B jsou po řadě hodnoty červeného, zeleného a modrého kanálu [7].

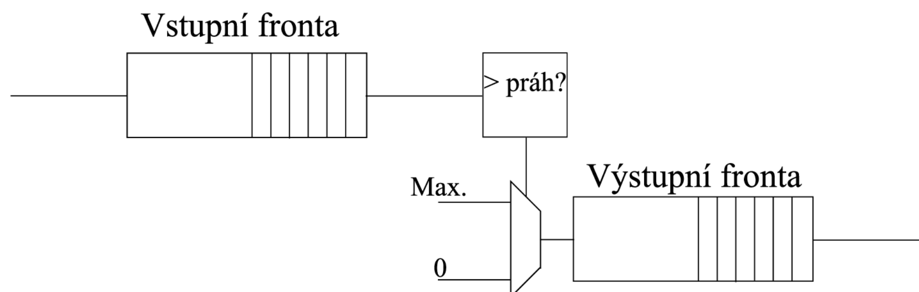
Pro různé účely vzniklo několik různých modifikací prahového filtru. Náhodným generováním prahu pro každý bod dosáhneme efektu "hrubého zrna", který simuluje vzhled starých fotografií. Při detekci několika objektů v jednom obraze se často používá adaptivní prahování, které pro každou část obrazu používá jiný práh. Ten je často automaticky odvozován, například z histogramu obrazu. Jinou variantou je prahový filtr pracující se dvěma prahy. Zda bod patří objektu se rozhodne podle toho, zda leží vně nebo uvnitř intervalu ohraničeného zadanými prahy [10, 7].

$$I = 0,299R + 0,587G + 0,114B \tag{3.1}$$

$$f(x) = \begin{cases} 0 & \text{pro } x < T \\ 1 & \text{pro } x \geq T \end{cases} \tag{3.2}$$

Základní varianta filtru, jejíž implementace je zde srovnávána, pracuje tak, že každý bod porovná s předem zvoleným prahem a pokud je hodnota bodu menší než zvolený práh,

je nová hodnota bodu 0 (nejnižší hodnota použitého rozsahu), bodu je přiřazena černá barva. Pokud je hodnota větší nebo rovna zadanému prahu, je bodu přiřazena hodnota 1 (nejvyšší hodnota použitého rozsahu), bod je zobrazován bílou barvou. Podle interpretace jsou pak v obraze bílé objekty obklopeny černým okolím nebo naopak. Funkční předpis filtru je uveden ve vzorci 3.2, kde x je hodnota aktuálně zpracovávaného bodu a T je hodnota zvoleného prahu. Schematicky je filtr zakreslen na obrázku 3.1. Skládá se ze dvou



Obrázek 3.1: Schéma prahového filtru

front, vstupní a výstupní, komparátoru a multiplexoru. Nově příchozí bod je umístěn do vstupní fronty. Ve chvíli, kdy je bod na řadě, je přiveden na vstup komparátoru, který jej porovná s konstantním prahem a ovládá multiplexor, který na základě výsledku porovnání do výstupní fronty vloží minumum či maximum zvoleného rozsahu, odtud je pak hodnota vyčtena jako výsledek filtrování. Pokud je při implementaci front použita paměť schopná v jednom taktu zároveň čtení i zápisu, bude filtr každý hodinový takt produkovat jeden bod.

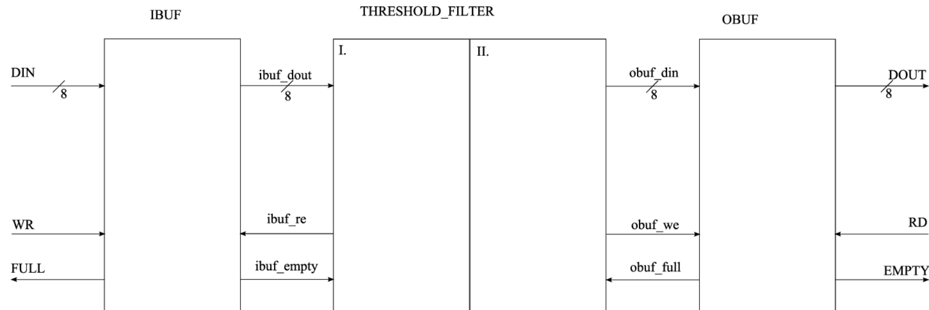
3.1.1 VHDL

Pro implementaci ve VHDL bylo potřeba vytvořit entitu `dual_port_fifo`, která realizuje frontu s odděleným zápisovým a čtecím rozhraním. V ukázce kódu 3.1 je zobrazena definice entity.

```
1  entity dual_port_fifo is
2    generic (
3      DEPTH        : positive;
4      WIDTH        : positive;
5      N_FULL       : integer := 0
6    );
7    port (
8      FULL_PLUSN   : buffer std_logic;
9      EMPTY        : buffer std_logic;
10     RESET        : in std_logic;
11     CLKW         : in std_logic;
12     WE           : in std_logic;
13     DIN          : in std_logic_vector(WIDTH-1 downto 0);
14     CLKR         : in std_logic;
15     RE           : in std_logic;
16     DOUT         : out std_logic_vector(WIDTH-1 downto 0)
17   );
18 end entity dual_port_fifo;
```

Ukázka kódu 3.1: Entita dual_port_fifo

Paramety fronty je možné upravit specifikací generických parametů. DEPTH určuje počet položek, které je možné do fronty uložit a musí být mocninou dvou. Parametrem WIDTH je možné určit počet bitů jedné položky, N_FULL udává počet volných pozic ve frontě ve chvíli, kdy je signál FULL_PLUSN uveden do aktivní úrovně a fronta je prohlášena za plnou. Této vlastnosti je využito při zřetězeném zpracování, kdy je třeba počítat s tím, že když v prvním kroku ověříme obsazenost fronty v druhém kroku může být fronta již obsazena. Signál EMPTY indikuje nulovou obsazenost fronty. Každému rozhraní může být poskytován samostatný hodinový signál a jedná se tedy o asynchronní implementaci fronty. Nicméně asynchronnosti není v tomto případě využito. Tato entita je využita jako vstupní a výstupní fronta.



Obrázek 3.2: Realizace prahového filtru ve VHDL

Jádrem entity je proces THRESHOLD_FILTER, který pracuje ve dvou krocích, které jsou zřetězeny a při dostatečném přísunu a odběru hodnot do vstupní respektive z výstupní fronty je entita schopna každý takt hodinového signálu vyprodukovat jednu hodnotu. V prvním kroku je ověřena dostatečná kapacita výstupní fronty a neprázdnost vstupní fronty, pokud jsou tyto požadavky splněny, je aktivován signál ibuf_re, na základě kterého v dalším taktu vstupní fronta na výstup vystaví aktuální hodnotu. Zároveň je povolen druhý krok zřetězení, ve kterém se provede porovnání hodnoty vyčtené ze vstupní fronty s prahem a je generována odpovídající hodnota a signál obuf_we, který způsobí uložení hodnoty do výstupní fronty. Na obrázku 3.2 je znázorněna struktura entity a napojení rozhraní na fronty.

3.1.2 Vysokoúrovňový nástroj

Implementace s využitím vysokoúrovňového nástroje byla provedena v jazyce C++ rozšířeném o datové typy z knihovny *Algoritmic C Datatype*. Jedná se o knihovnu vyvinutou společností Mentor Graphics© obsahující definici tříd umožňujících v prostředí jazyka C++ pracovat s bitovou přesností. Obsahuje třídy pro reprezentaci celých čísel, čísel v pevné řádové čárce a komplexních čísel. Umožňuje výběr znaménkové či bezznaménkové varianty [8].

```
1 #pragma hls_design top
2 void treshold(bool we, ac_int<8,false> din, bool re, ac_int<8,false> &dout,
      bool &empty, bool &full) {
3
4         IBUF_FILL: if (we && ibuf_cnt < IBUF_SIZE) {
5                 ibuf[ibuf_wr++] = din;
6                 ++ibuf_cnt;
7         }
8         OBUF_READ: if (re && obuf_cnt > 0) {
```

```
 9                  dout = obuf[obuf_rd++];
10                  --obuf_cnt;
11          }
12          TRESHOLD_FILTER: if (ibuf_cnt > 0 && obuf_cnt < OBUF_SIZE) {
13                  obuf[obuf_wr++] = (ibuf[ibuf_rd++] > 127 ? 255 : 0);
14                  ++obuf_cnt; --ibuf_cnt;
15          }
16          empty = (obuf_cnt = 0);
17          full = (ibuf_cnt = IBUF_SIZE);
18  }
```

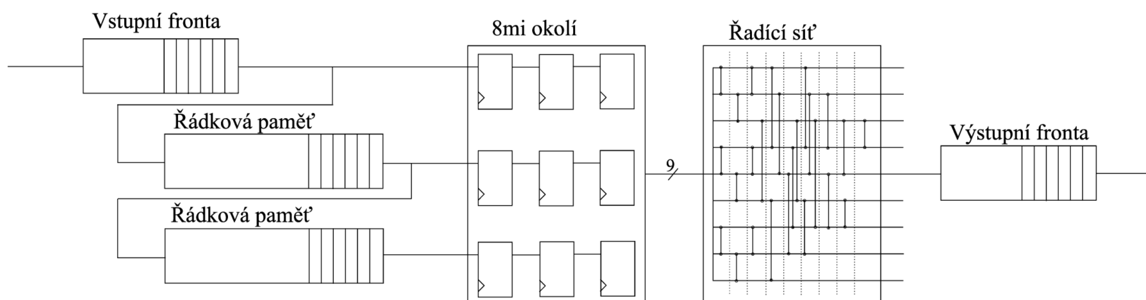
Ukázka kódu 3.2: Implementace prahového filtru v C++

Ukázka kódu 3.2 obsahuje popis prahového filtru, pouze byly pro stručnost vynechány definice proměnných a konstant. Kód se skládá ze čtyř nezávislých částí. První, označená návěštím `IBUF_FILL` (řádky 4–7), řídí ukládání vstupních hodnot do vstupní fronty, druhá, označená `OBUF_READ` (řádky 5–8), obsluhuje vystavování hodnot z výstupní fronty na výstup komponenty. Třetí, značená návěštím `THRESHOLD_FILTER` (řádky 9–12), načítá hodnoty ze vstupní fronty a na základě porovnání vkládá do výstupní fronty hodnotu 0 nebo 255 (maximum osmibitového rozsahu). Poslední dva řádky generují signály `empty` a `full`, které signalizují plnou obsazenost vstupní fronty resp. nulovou obsazenost výstupní fronty. Návěští slouží pouze pro lepší orientaci v jednotlivých částech komponenty. Část uvozená návěštím `THRESHOLD_FILTER` byla následně zřetězena tak, aby byla komponenta schopna produkovat v každém taktu novou hodnotu. Pro správnou funkčnost je nutné doplnit definice proměnných a v závislosti na velikosti front zvolit, zda budou mapovány do registrových polí či do blokových pamětí dostupných v FPGA čipu.

3.2 Mediánový filtr

Mediánové filtrování je nelineární nízkofrekvenční filtrovací metoda, která se používá pro odstranění šumu typu „sůl a pepř“ z obrazu. Mediánový filtr překonává lineární nízkofrekvenční filtry na tomto typu šumu, protože potenciálně dokáže odstranit veškerý šum bez ovlivnění nezašuměných pixelů. Mediánový filtr odstraňuje samostatné pixely nezávisle na tom, jestli jsou světlé nebo tmavé. Hlavní nevýhodou filtru je možnost zničení detailů v obraze. Jako mnoho dalších nízkoúrovňových algoritmů pro zpracování obrazu je i mediánový filtr velmi jednoduchý.

Úskalím implementace filtru je potřeba pro každý bod seřadit pole o velikosti zvoleného okolí bodu. U základní varianty je používáno osmiokolí a je tedy potřeba seřadit 9 hodnot. Po seřazení okolních bodů je jediným zbývajícím krokem vyběr prostřední hodnoty (mediánu) a jeho propagace na výstup [3]. Proto je vhodné se soustředit na způsob řazení. Při implementaci běžných řadících algoritmů v hardware vzniká komplikovaná síť multiplexorů, která vnáší do řazení zpoždění, navíc běžné algoritmy často nedovolují jednoduše paralelizovat průběh řazení. Proto se pro implementaci řazení v hardware osvědčily řadící sítě, které využívají paralelní povahy hardware. Řadící sítě jsou vhodné pro řazení předem známého počtu prvků. Skládají se z operátorů typu „porovnej a vyměň“ (compare-and-swap), které porovnají dvě čísla na vstupu a v případě potřeby je prohodí tak, aby na jednom výstupu bylo vždy maximum a na druhém minimum ze vstupních čísel. Výhodou je, že datově nezávislé operace mohou probíhat paralelně a síť lze rozdělit do jednotlivých kroků a aplikovat zřetězení [2].



Obrázek 3.3: Schéma mediánového filtru

3.2.1 VHDL

Vstupní a výstupní fronta filtru je realizována entitou `dual_port_fifo`, která je popsána výše u prahového filtru. Dále byla vytvořena entita `dual_port_pipe`, která realizuje zpoždění vložené hodnoty o genericky nastavitelný počet hodnot. Tato entita slouží jako řádková paměť. Na obrázku 3.3 je zobrazeno schéma filtru. Data jsou vkládány do vstupní fronty, odkud jsou vyčítány do registrového pole, realizujícího osmiokolí. Vždy, když je hodnota vyčtena ze vstupní fronty nebo první řádkové paměti, je vložena do následující řádkové paměti, tím je zajištěno uchování posledních tří zpracovávaných řádků a jejich použití pro výpočet mediánu.

Jádrem filtru je devítivstupá řadící síť, která je implemetována jako samostatná entita. Rozhraní tvoří 9 osmibitových vstupů, 9 osmibitových výstupů, signál povolující řazení a signál indikující platná data na výstupu. Síť produkuje na výstupu seřazená vstupní data po 9 taktech hodinového signálu. Je však využito zřetězení a při dostatečném přísunu dat na vstup získáváme seřazenou posloupnost s každým hodinovým taktem. Tohoto zřetězení je dosaženo vložením devíti registrů mezi každý porovnávací stupeň sítě.

Prostřední z výstupů sítě je připojen na vstup výstupní fronty a signál potvrzující platná data na výstupu sítě je připojen na port povolení zápisu do výstupní fronty.

3.2.2 Vysokoúrovňový nástroj

Implementace pomocí vysokoúrovňového nástroje je velice podobná jako u prahového filtru. Načítání hodnot ze vstupu i vystavování na výstup je realizováno totožným způsobem. Navíc jsou použity dvě pole realizující řádkové paměti a pole o devíti prvcích, které slouží jako osmiokolí. Nejpodstatnější rozdíl je v implementaci vlastního filtru, kde je volána funkce `median`, která implementuje pomocí sekvence porovnání a výměn řadící síť a vrací prostřední prvek. Ten je pak vložen do výstupní fronty, odkud může být vyčten.

3.3 Klasifikační jednotka

Při zpracování paketů v routerech, firewallech a jiných síťových prvcích jsou procházející pakety děleny do kategorií (toků). Každý tok charakterizuje množina pravidel, které každý paket patřící do tohoto toku musí splňovat. Například všechny pakety se stejnou zdrojovou a cílovou adresou mohou tvořit tok. Síťový prvek pak se všemi pakety z jednoho toku nakládá stejným způsobem. Při aplikaci zpracovávání toků na routerech obsluhujících vysokorychlostní linky je nutné provádět klasifikaci paketů v hardware. Router musí totiž

provádět několik činností zároveň. Každý paket totiž typicky prochází filtrováním na vstupním rozhraní, výpočtem výstupního rozhraní a filtrováním na výstupním rozhraní. Tyto tři úkoly jsou časově náročné a při množství paketů, které je třeba obsloužit, by nebylo v silách běžného procesoru je zvládnout. Za tímto účelem vzniklo množství algoritmů pro klasifikaci paketů v hardware [6].

Základní verze klasifikátoru, která byla implementována se skládá, z paměti uchovávající klasifikační pravidla, série porovnávání a pomocné logiky pro vyčítaní paměti a vyhodnocování porovnání. Do paměti jsou vloženy kritéria pravidel a při porovnání jsou postupně porovnávána všechna uložená pravidla. Poslední splněné pravidlo je po skončení porovnávání předáno jako výsledek. Každé pravidlo je uloženo jako struktura v ukázce kódu 3.3, proměnné `sip` a `dip` uchovávají zdrojovou a cílovou IP adresu. V proměnných `smask` a `dmask` jsou uloženy masky zdrojové a cílové IP adresy. `Sport` a `dport` reprezentují zdrojový a cílový port TCP či UDP datagramu. `Protocol` uchovává typ přenášeného protokolu a `control` slouží ke globálnímu povolení pravidla a povolení jednotlivých záznamů.

```
1 struct rule{
2     ac_int<32,false> sip, dip;
3     ac_int<32,false> smask, dmask;
4     ac_int<16,false> sport, dport, protocol;
5     ac_int<6,false> control;
6 };
```

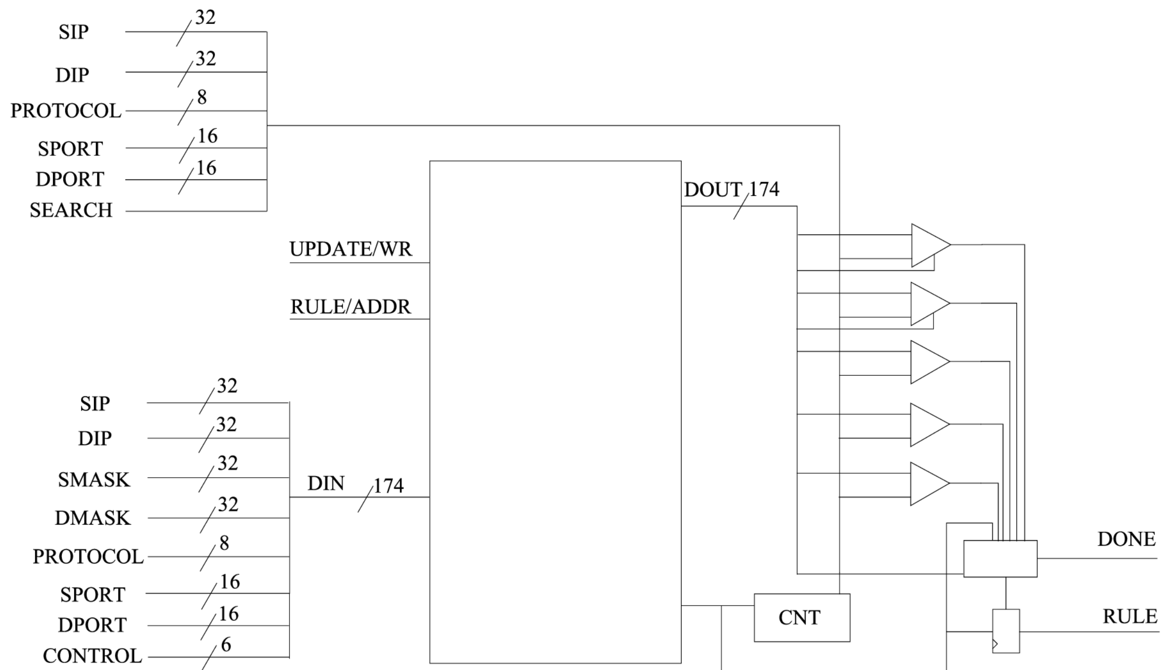
Ukázka kódu 3.3: Struktura pravidla

Abychom byli schopni dosáhnout plné rychlosti linky, musíme porovnání zvládnout nejdéle za stejný čas, jaký trvá přenos nejkratšího možného paketu. Pro protokol IPv4 je to 64 B [1]. To tedy znamená, že klasifikace nesmí zabrat delsí čas než přenos 512 b. Tento údaj je ale závislý na rychlosti linky, se kterou pracujeme. Pokud máme k dispozici linku o rychlosti 1 Gbps je to 512 ns, pro 10 Gbps je to už jen 51,2 ns. Jedná se tedy o klasifikaci téměř 2 resp. 20 milionů paketu za sekundu. Při reálných rycholostech FGPA čipu okolo 200Mhz bychom mohli porovnávat maximálně 100 resp. 10 pravidel, avšak můžeme vytvořit jednotku, která bude porovnávat takto malý počet pravidel a umístit ji do výsledného čipu mnohokrát vedle sebe, tím získáme počet pravidel omezený jen zdroji použitého FPGA čipu.

3.3.1 VHDL

Na obrázku 3.4 je znázorněno schéma klasifikační jednotky implementované ve VHDL. Paměť je realizována entitou `dual_port_ram`, ta implementuje oddělený zápisový a čtecí port. Zápisovým rozhraním jsou do paměti vkládány klasifikační kritéria pro pravidlo, které je přivedeno na adresový vstup. Přidání nového pravidla či aktualizace existujícího pravidla je povolena přivedením log. 1 na signál `UPDATE`.

Vyhledání pravidla v paměti pak probíhá sekvenčním vyčítaním jednotlivých kritérií a jejich srovnáváním s předanými údaji. Součástí uložených dat je i kontrolní vektor, který udává, zda je pravidlo aktivní a které údaje se mají porovnávat. Během porovnávání se propaguje na výstup vždy nejvyšší aktuálně nalezené pravidlo a lze tak sledovat všechna pravidla, kterým analyzovaný paket odpovídá. Poté, co je porovnávání dokončeno, je generován signál `DONE` a na výstupu je nejvyšší pravidlo, kterému paket odpovídá. Tento stav setrvává, dokud není opět aktivován signál `SEARCH`.



Obrázek 3.4: Schéma klasifikační jednotky

3.3.2 Vysokoúrovňový nástroj

Implementace za pomoci vysokoúrovňového nástroje se skládá ze statického pole struktur 3.3, udržujícího zadaná pravidla a dvou podmíněných bloků. První z nich umožňuje přidání či aktualizaci pravidla, druhý pak obsahuje smyčku, ve které jsou jednotlivé pravidla čteny a porovnávány s předanými údaji. Po skončení smyčky je generován potvrzovací signál.

Kapitola 4

Dosažené výsledky

V předchozí kapitole byly popsány způsoby implementace jednotlivých algoritmů oběma přístupy, nyní budou srovnány výsledky syntézy těchto implementací.

4.1 Prahový filtr

Prahový filtr byl prvním algoritmem, který byl implementován a sloužil tak autorovi této práce k seznámení s možnostmi vysokoúrovňového nástroje. Z toho důvodu trvala implementace vysokoúrovňovým nástrojem přibližně dvakrát více času než implementace ve VHDL. Dalším důvodem pro tento neočekávaný výsledek jsou i poměrně bohaté zkušenosti autora s jazykem VHDL. V tabulce 4.1 jsou uvedena množství zabraných zdrojů čipu a maximální dosažitelná frekvence.

	VHDL		Vysokoúrovňový nástroj	
	absolutně	relativně (%)	absolutně	relativně (%)
LUTs	127	8,27	349	22,72
CLB Slices	64	8,33	175	22,79
Dffs or Latches	50	2,62	204	10,69
Block RAMs	2	50	2	50
Maximální frekvence (MHz)	100,888	–	68,432	–

Tabulka 4.1: Výsledky syntézy prahového filtru

U tohoto algoritmu byl nízkoúrovňový popis velmi efektivní a překonal vysokoúrovňový nástroj úsporností zdrojů i maximální dosažitelnou frekvencí. Na obrázcích B.1 a B.2 v příloze můžeme porovnat RTL schémata vzniklá syntézou obou implementací. Na první pohled vidíme, že VHDL implementace vede na mnohem jednodušší RTL schéma. Ve schématu lze rovněž rozpoznat podobnost s obrázky 3.1 a 3.2, toto je způsobeno nízkoúrovňovým přístupem a plnou kontrolou nad výslednou architekturou v jazyce VHDL. Naproti tomu automatickým odvozováním z kódu ve vyšším programovacím jazyce vysokoúrovňovým syntézním nástrojem, vzniklo schéma, které vypadá dosti obecně a nic nenapovídá o výsledné funkci komponenty.

4.2 Mediánový filtr

Při implementaci mediánového filtru se již částečně projevila autorova znalost prostředí vysokoúrovňového nástroje a implementační čas v jazyce VHDL byl asi o třetinu až polovinu delší než ve vysokoúrovňovém nástroji. Značnou úsporu času představuje například možnost libovolně volit stupeň zřetězení pouze nastavením příslušného parametru, zatímco ve VHDL to znamená značně modifikovat popis. Taktéž rozbalování smyček lze realizovat podobným způsobem.

V tabulce 4.2 jsou uvedena množství potřebných zdrojů na čipu a maximální frekvence hodinového signálu na které může komponenta pracovat. U tohoto algoritmu se vysokoúrovňová syntéza ukázala jako efektivnější. Rozdíl však je pouze v jednotkách procent. Avšak maximální pracovní frekvence je opět nižší než u VHDL popisu, to naznačuje, že komponenta bude obsahovat složitější kombinační logiku, která prodlužuje cestu signálu mezi registry.

	VHDL		Vysokoúrovňový nástroj	
	absolutně	relativně (%)	absolutně	relativně (%)
LUTs	548	35,68	518	33,72
CLB Slices	274	35,68	259	33,72
Dffs or Latches	487	25,52	346	18,13
Block RAMs	4	100,00	4	100,00
Maximální frekvence (MHz)	83,077	–	62,235	–

Tabulka 4.2: Výsledky syntézy mediánového filtru

Na obrázcích B.3 a B.4 můžeme vidět RTL schémata, která jsou výsledkem syntézy VHDL popisu resp. vysokoúrovňového popisu. Na schématu B.3 je dobře patrná struktura komponenty a výsledné RTL je velice podobné obrázku 3.3, to způsobuje nízkoúrovňový popis ve VHDL a explicitní dělení komponenty na entity realizující jednotlivé funkční bloky. Na schématu B.4 jsou zřetelné vstupní, výstupní a řádkové paměti, zbylá logika je však zapouzdřena a struktura komponenty je nerozpoznatelná.

4.3 Klasifikační jednotka

Během implementace klasifikační jednotky se zcela projevila hlavní výhoda vysokoúrovňového nástroje. Implementace v jazyce VHDL zabrala asi dvakrát více času než implementace vysokoúrovňovým přístupem. Ovšem výsledky syntézy uvedené v tabulce 4.3 ukazují, že daní za rychlost implementace je dosti výrazný nárůst potřebných zdrojů.

	VHDL		Vysokoúrovňový nástroj	
	absolutně	relativně (%)	absolutně	relativně (%)
LUTs	356	0,37	1236	1,27
CLB Slices	136	0,56	322	1,32
Dffs or Latches	544	0,56	1287	1,32
Block RAMs	3	1,42	5	2,36
Maximální frekvence (MHz)	264,76	–	169,75	–

Tabulka 4.3: Výsledky syntézy klasifikátoru

RTL schémata nejsou vzhledem k jejich velikosti uvedena přímo v přílohách práce, ale lze je nalézt na přiloženém CD. Ve schématu vzniklém syntézou vysokoúrovňového popisu je viditelné, že data jsou uložena do více samostatných paměťových bloků. Samotná porovnávací a vyhodnocovací logika je zapouzdřena a není patrná. Naproti tomu RTL schéma vzniklé syntézou VHDL popisu obsahuje velké množství porovnávacích prvků a pouze jeden paměťový blok.

Kapitola 5

Závěr

Cílem této práce bylo srovnání výsledků dvou odlišných přístupů k tvorbě hardwarových komponent. Jedním ze srovnávaných přístupů je dnes nejčastěji používaný nízkorúrovňový popis, kdy je zcela na vývojáři, jak naloží s dostupnými zdroji a jak bude cílová architektura vypadat. Druhým způsobem je použití vysokoúrovňového nástroje. Přechod k tomuto přístupu je s rostoucí složitostí navrhovaných komponent jen otázkou času. Při vysokoúrovňové syntéze je však komponenta popisována na vyšší úrovni abstrakce, a výsledná architektura a přesná struktura je před vývojářem skryta. Mnohá implementační rozhodnutí jsou automaticky odvozena nástrojem. Je proto vhodné srovnat tyto přístupy, abychom měli přehled o ceně, kterou za zvyšování abstrakce platíme.

Obecně lze říci, že vývoj pomoci nízkoúrovňového popisu v jazyce VHDL je efektivnější co do použití zdrojů a zároveň vede na komponenty pracující na vyšší frekvenci hodinového signálu. Avšak vysokoúroňový nástroj nabízí výrazné zkrácení doby vývoje a integrací verifikačních nástrojů a propojením se simulačními a dalšími nástroji zvyšuje komfort práce. Volba nízko- či vysoko- úrovňového přístupu je na zvážení vývojáře na základě požadavků kladeného na konkrétní komponentu. U aplikací, kde je doba vývoje důležitější než optimálnost a efektivita výsledného produktu, je vysokoúrovňový přístup vhodný.

V této práci je možné pokračovat například rozšířením vzorku testovaných algoritmů, případně jejich sofistikovanějším výběrem. Alternativou je prozkoumání všech možností, které vysokoúrovňový syntézní nástroj integruje. Zajímavé by mohlo také být vytvoření komplexního systému skládajícího se z více komponent popsaného v tomto nástroji, tuto možnost však verze, která byla použita, nenabízí.

Literatura

[1] *INTERNET PROTOCOL* [RFC 791]. září 1980. Dostupné na: <http://www.ietf.org/rfc/rfc791.txt>.

[2] Ajtai, M., Komlós, J. a Szemerédi, E. An 0(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1983. S. 1–9. STOC '83. ISBN 0-89791-099-0.

[3] Dhanasekaran, D., Krishnamurthy, A. a Ramkumar, J. High speed pipeline architecture for adaptive median filter. In *Proceedings of the International Conference on Advances in Computing, Communication and Control*. New York, NY, USA: ACM, 2009. S. 597–600. ICAC3 '09. ISBN 978-1-60558-351-8.

[4] Fingeroff, M. *High-Level Synthesis Blue Book*. [b.m.]: Xlibris Corporation, 2010. ISBN 1450097243.

[5] Gajski, D. D., Dutt, N. D., Wu, A. C.-H. et al. *High-Level Synthesis: Introduction to Chip and System Design*. [b.m.]: Springer, 1992. ISBN 0792391942.

[6] Gupta, P. a McKeown, N. Algorithms for packet classification. *Network, IEEE*. Mar/apr 2001, roč. 15, č. 2. S. 24 –32. ISSN 0890-8044.

[7] Kršek, P. a Španěl, M. *Základy počítačové grafiky: Redukce barevného prostoru*. 2011.

[8] Martínek, T. *Pokročilé číslicové systémy: Catapult C*. 2010.

[9] Martínek, T. *Pokročilé číslicové systémy: Pokročilé metody syntézy číslicových obvodů*. 2010.

[10] Sezgin, M. a Sankur, B. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*. 2004, roč. 13, č. 1. S. 146–168.

Příloha A

Obsah CD

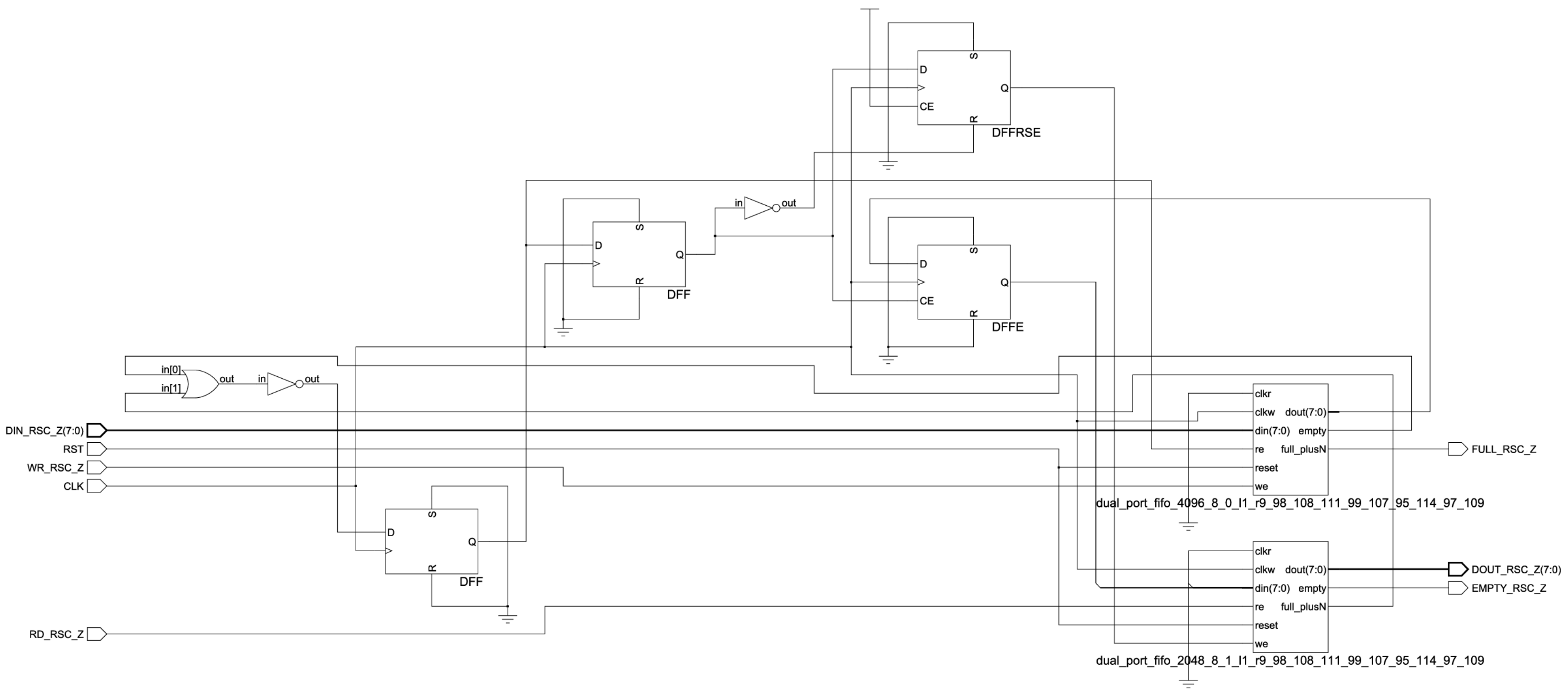
`latex/` Adresář obsahující zdrojové kódy této bakalářské prace v přeložitelné podobě.

`designs/` V tomto adresáři se nacházejí testované algoritmy v následující struktuře:

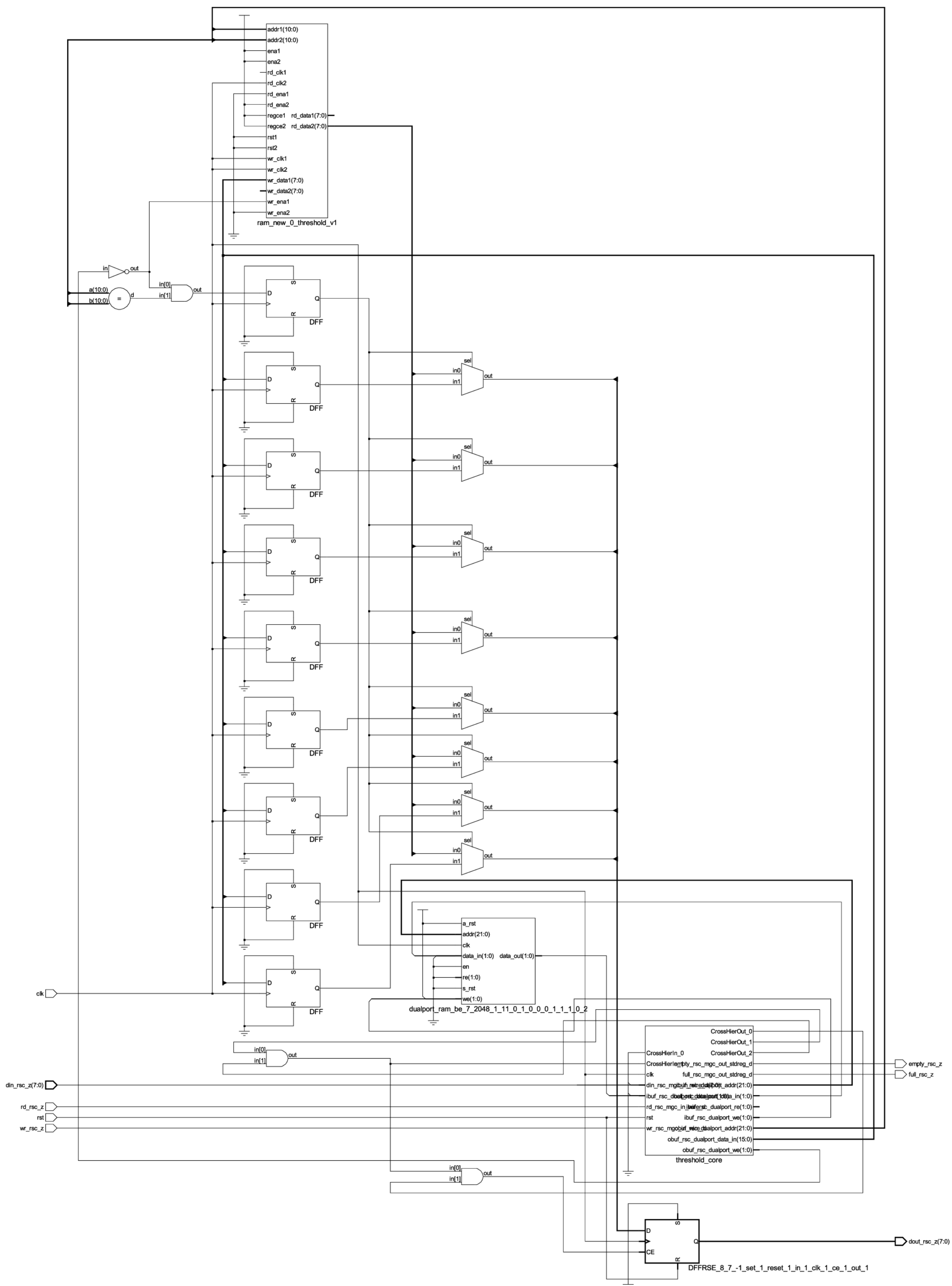
- `Median/` Mediánový filtr.
  - `HighLevel/` Adresář obsahující soubory vysokoúrovňové syntézy.
  - `vhdl/` Adresář obsahující soubory VHDL popisu.
  - `sim/` Adresář obsahující soubory pro simulaci.
- `Threshold/` Prahový filtr.
  - `HighLevel/` Adresář obsahující soubory vysokoúrovňové syntézy.
  - `vhdl/` Adresář obsahující soubory VHDL popisu.
  - `sim/` Adresář obsahující soubory pro simulaci.
- `Classification/` Klasifikační jednotka.
  - `HighLevel/` Adresář obsahující soubory vysokoúrovňové syntézy.
  - `vhdl/` Adresář obsahující soubory VHDL popisu.
  - `sim/` Adresář obsahující soubory pro simulaci.
- `shared/` Adresář obsahující soubory společné pro více implementací.

Příloha B

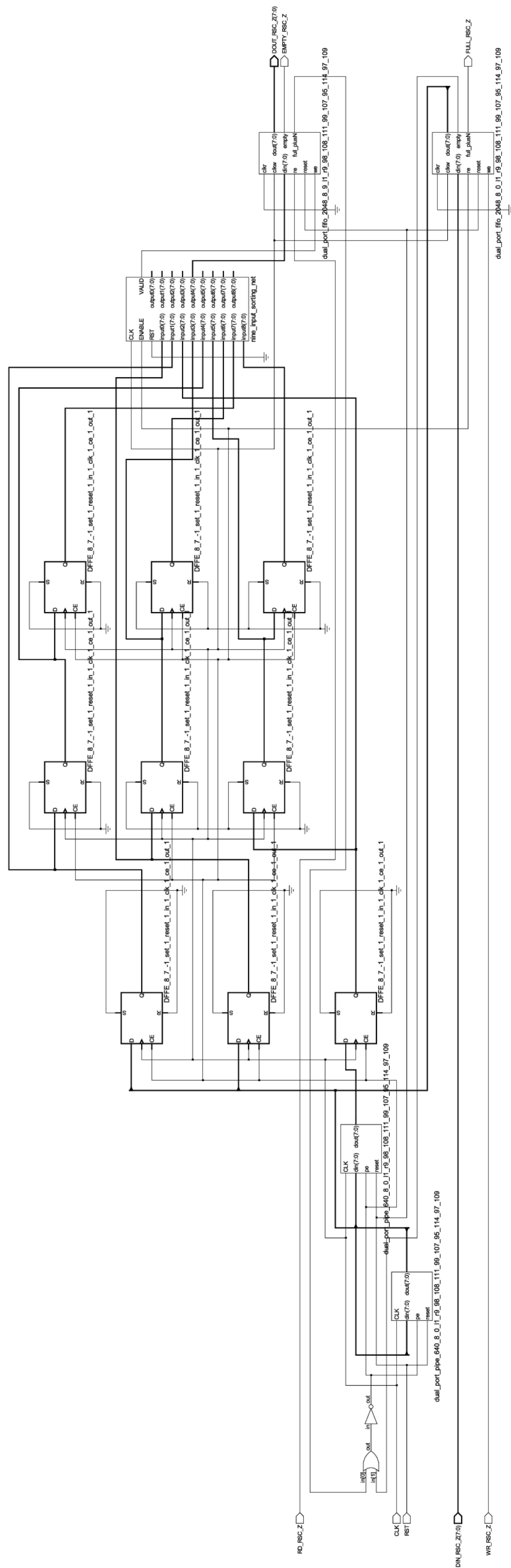
RTL schémata



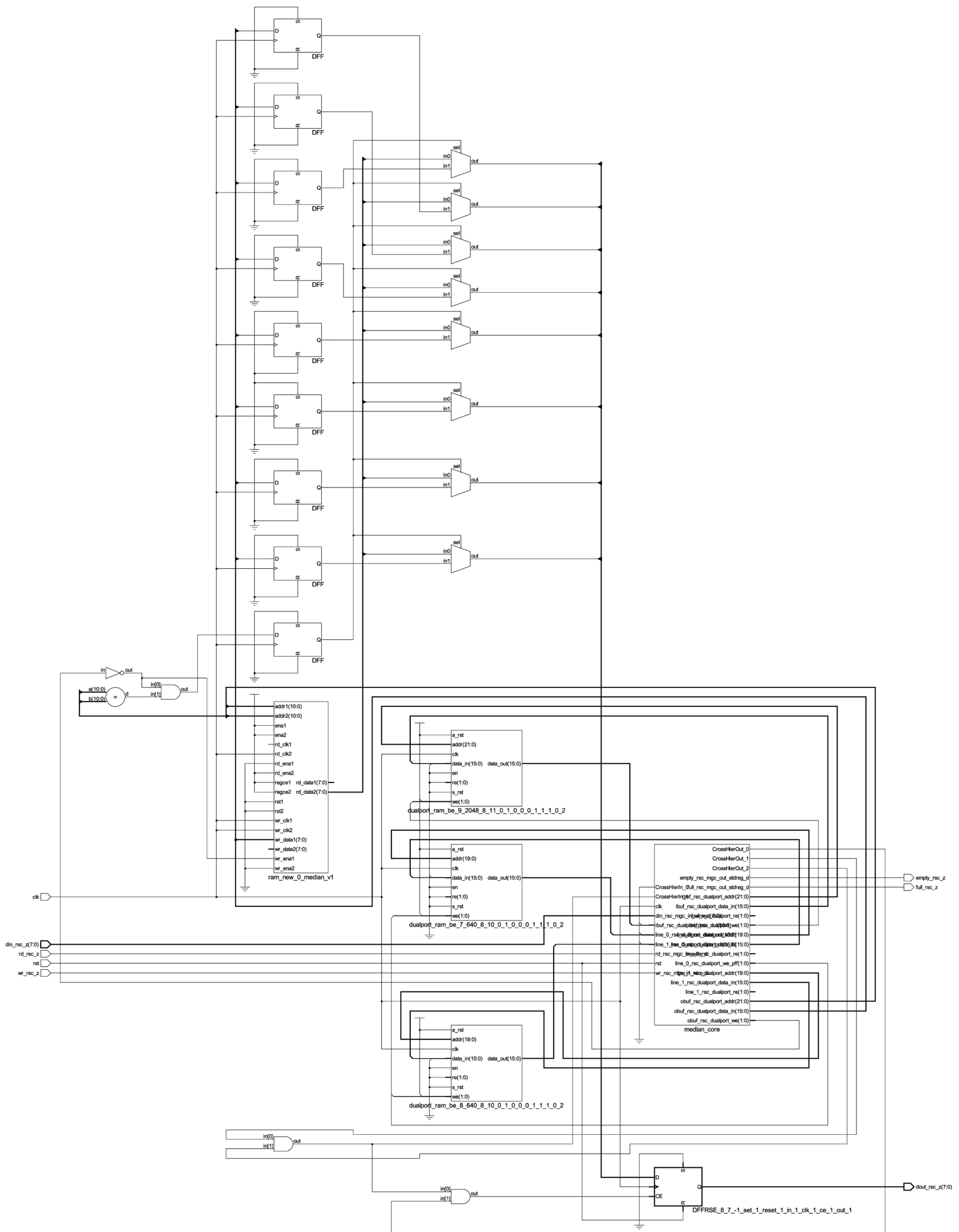
Obrázek B.1: Schéma prahového filtru získaného syntézou VHDL popisu



Obrázek B.2: Schéma prahového filtru získaného vysokoúrovňovou syntézou



Obrázek B.3: Schéma mediánového filtru získaného syntézou VHDL popisu



Obrázek B.4: Schéma mediánového filtru získaného vysokoúrovňovou syntézou