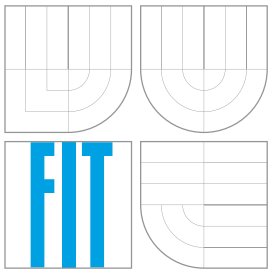
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AKCELERACE HDR TONE–MAPPINGU NA PLATFORMĚ XILINX ZYNQ

HDR TONE–MAPPING ACCELERATION ON XILINX ZYNQ PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. SVETOZÁR NOSKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN MUSIL

BRNO 2016

Zadání diplomové práce/17540/2015/xnosko01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Nosko Svetozár, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Akcelerace HDR tone-mappingu na platformě Xilinx Zynq**
HDR Tone-Mapping Acceleration on Xilinx Zynq Platform
Kategorie: Vestavěné systémy

Pokyny:

1. Prostudujte problematiku syntézy na systémové úrovni (HLS) pro obvody FPGA. Seznamte se s vývojovým prostředím Vivado HLS a platformou ZYNQ firmy Xilinx.
2. Prostudujte problematiku pořizování a zpracování HDR obrazu, zaměřte se zejména na tzv. tone-mapping metody.
3. Vybrané metody tone-mappingu implementujte v prostředí Vivado HLS pro platformu Xilinx Zynq.
4. Syntetizujte implementovaný obvod pro jeden z vývojových kitů s čipem Xilinx Zynq a ověřte funkčnost na reálné aplikaci.
5. Zhodnoťte výsledky práce a diskutujte případné pokračování nebo rozšíření práce.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Musil Martin, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato diplomová práca je zameraná predovšetkým na syntézu na systémovej úrovni (HLS). Prvá časť obsahuje teoretické detaily a postupy, ktoré sa využívajú v HLS nástrojoch. Ďalej nasleduje popis syntézy v nástroji Vivado HLS, ktorý je využitý pri implementácii aplikácie. Druhá časť obsahuje potrebné teoretické poznatky z oblasti obrazu s vysokým dynamickým rozsahom a mapovania tónov. Tretia časť je venovaná návrhu a implementácií aplikácie, ktorá realizuje metódy mapovania tónov v HDR snímkach. Vybrané metódy sú implementované vo Vivado HLS a jazyku C++. Táto aplikácia je postavená na platforme Xilinx Zynq a využíva multiexpozičnú kameru pre záznam snímok HDR. Snímky sú predané do FPGA na spracovanie, kde prebieha mapovanie tónov.

Abstract

This diploma thesis focuses on the High-level synthesis (HLS). The first part deals with theoretical details and methods that are used in HLS tools. This is followed by a description of the synthesis tool Vivado HLS which will be used for implementation of an application. In the second part is briefly introduced high dynamic range images (HDR) and tone mapping. The third part is dedicated to design and implementation of the aplication which implements tone mapping methods in HDR images. This methods are implemented in Vivado HLS and language C++. This application is based on platform Xilinx Zynq and it uses multiexposure camera for capturing HDR images. Images are transmitted to FPGA for tone mapping processing.

Kľúčové slová

Xilinx Zynq, SoC, HLS, High Level Synthesis, Vivado, Vivado HLS, HDR, mapovanie tónov

Keywords

Xilinx Zynq, SoC, HLS, High Level Synthesis, Vivado, Vivado HLS, HDR, tone mapping

Citácia

NOSKO, Svetozár. *Akcelerace HDR tone–mappingu na platformě Xilinx Zynq*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Musil Martin.

Akcelerace HDR tone–mappingu na platformě Xilinx Zynq

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Musila. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Svetozár Nosko
23. mája 2016

Poďakovanie

Ďakujem svojmu vedúcemu Ing. Martinovi Musilovi za odborné vedenie a podnety, ktoré mi poskytol.

Obsah

1 Úvod **2**

2 Syntéza na systémovej úrovni **4**
2.1 Dôvody pre Syntézu na systémovej úrovni 4
2.2 Syntéza na systémovej úrovni 6
2.3 Vivado High-Level Synthesis 14
2.4 Programovací model Vivado HLS 17
2.5 Zynq-7000 All Programmable SoC 25

3 Snímky s vysokým dynamickým rozsahom **33**
3.1 Problematika HDR 33
3.2 Mapovanie tónov 37
3.3 Dragov globálny operátor 38
3.4 Reinhardov operátor 40
3.5 Lokálny operátor Durand a Dorsey 42

4 Špecifikácia zadania **44**
4.1 Stanovenie cieľov 44

5 Implementácia **46**
5.1 Návrh aplikácie 46
5.2 Dragov operátor 47
5.3 Reinhardov operátor 50
5.4 Lokálny operátor Durand a Dorsey 52
5.5 Optimalizácie operátorov 56
5.6 Realizácia aplikácie na Xilinx Zynq 60

6 Záver **63**

Literatúra **64**

Prílohy **66**
Zoznam príloh 67

A Obsah CD **68**

B Ukážka bilaterálneho filtrovania **69**

C Schéma aplikácie **70**

Kapitola 1

Úvod

V posledných rokoch sa čoraz častejšie stretávame so systémami, ktoré vyžadujú spracovanie obrovského množstva dát v reálnom čase. Typickou oblasťou je oblasť spracovania signálov, vysokorýchlostných dátových tokov, obrazu a videa. Často nie je z hľadiska výkonu možné využiť procesor, ktorý síce poskytuje flexibilitu z hľadiska programovateľnosti a modifikácie programu, ale problémom je nízky výkon a vysoká spotreba v porovnaní s inými platformami. Dnešné technológie umožňujú tieto problémy riešiť. Možnosťou je napríklad využitie technológie ASIC (Application Specific Integrated Circuit) alebo FPGA (Field Programable Gate Array). Voľbou aplikačne špecifického obvodu dostaneme obvod, ktorý je navrhnutý špeciálne pre konkrétnu aplikáciu s obrovským výkonom, avšak za cenu dlhej doby návrhu, vývoja a nízkej flexibility využitia výsledného obvodu. Inou voľbou je využitie technológie FPGA, ktorá poskytuje vysoký výkon, určitú mieru flexibility a rýchly návrh obvodov s rýchlym uvedením na trh. Čoraz častejšie sa objavujú platformy, ktoré obsahujú moderný procesor, často na báze ARM-Cortex, ku ktorému je na jednej doske prostredníctvom zbernice pripojená programovateľná logika FPGA, ktorá poskytuje obrovské možnosti pre vytvorenie vlastných akceleračných jednotiek alebo dokonca procesorov. Takéto platformy poskytuje mnoho výrobcov ako Xilinx alebo Altera.

Realizácia týchto algoritmov v FPGA často môže byť veľmi rozsiahla a sú potrebné vhodné jazyky pre ich popis. Medzi najčastejšie používané patria VHDL a Verilog. Neustála platnosť Moorovho zákona so sebou prináša veľký nárast počtu tranzistorov na čipe, rozsahu a zložitosti navrhovaných aplikácií, ktorá núti k hľadaniu cesty, ako zvládnuť zvyšujúcu sa náročnosť takéhoto návrhu. Prirodzenou cestou je zvýšenie abstrakcie z popisu obvodov na popis pomocou jazykov vyššej úrovne, ako sú jazyky C a C++. Proces prevodu z takéhoto jazyka sa nazýva syntéza na systémovej úrovni. V priebehu vývoja vzniklo veľké množstvo nástrojov, ktoré umožňujú takýto prevod. Medzi najznámejšie patria Vivado HLS, CatapultC alebo HandledC.

Prvá časť práce sa teda orientuje na uvedenie do problematiky syntézy na systémovej úrovni a to prostredníctvom prostredia Vivado HLS. Druhá časť práce sa orientuje na praktické využitie možností HLS. Zvolenou je oblasť zo spracovania HDR obrazu a videa, konkrétne na tzv. mapovaní tónov (z anglického tone mapping). Mapovanie tónov je operácia pri spracovaní obrazu a videa, ktoré má vysoký dynamický rozsah. Tento rozsah je pomocou tejto operácie prevedený do rozsahu, ktorý je vhodný pre zobrazenie na displejoch či tlačiarňach, ktoré majú obmedzený dynamický rozsah. Mapovanie tónov teda napomáha zachovaniu kontrastu a detailov zobrazovanej scény.

V nasledujúcej kapitole je pozornosť venovaná problematike syntézy na systémovej úrovni. Najskôr je rozobraná problematika všeobecne a následne sa zameriava konkrétne na proces syntézy Vivado HLS. Ďalšou témou je samotný programovací model pre Vivado HLS, kde sú rozobrané základné postupy a možnosti návrhu obvodov pomocou vysokoúrovňových jazykov. Nakoniec je popísaná platforma Xilinx Zynq, ktorá bude využitá v praktickej časti. Tretia kapitola obsahuje úvod do problematiky HDR a mapovania tónov. Štvrtá kapitola popisuje stanovenie cieľov, za ktorou nasleduje kapitola venovaná implementácií. Poslednou kapitolou je Záver.

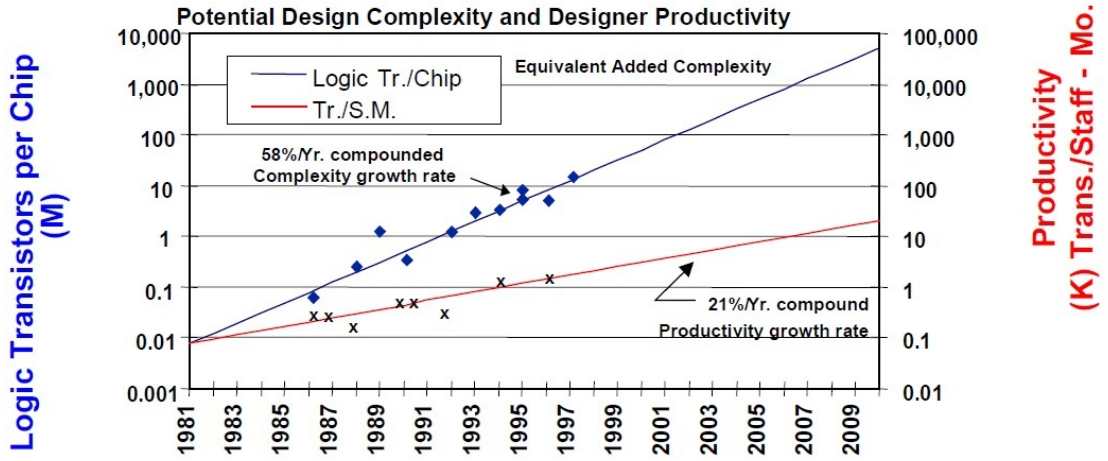
Kapitola 2

Syntéza na systémovej úrovni

V tejto kapitole je rozobraná problematika syntézy na systémovej úrovni. Najskôr sa zaoberá dôvodmi pre prechod na tento typ syntézy a následne na detaily tejto problematiky. Po všeobecnom úvode do syntézy na systémovej úrovni nasleduje popis tohoto procesu v prípade Vivado HLS. Ďalej sú rozobrané základy a špecifiká programovania pre proces syntézy. Na konci kapitoly sa nachádza popis architektúry Xilinx Zynq.

2.1 Dôvody pre Syntézu na systémovej úrovni

V roku 1965 spoluzakladateľ firmy Intel Gordon Moore vyslovil asi najznámejšie empirické pravidlo informatiky, ktoré hovorí: „Počet tranzistorov, ktoré môžu byť umiestnené na integrovaný obvod sa pri zachovaní rovnakej ceny zhruba každých 18 mesiacov zdvojnásobí“. Toto tvrdenie v roku 1975 upravil na dva roky a toto pravidlo platí dodnes. Avšak tieto tranzistory je potrebné využiť, každým zvýšením počtu tranzistorov sa zvyšuje aj rozsah navrhovaných obvodov a zložitosť narastá každým rokom o cca. 60%, ale produktivita práce návrhárov rastie iba o 20% [1]. Tento trend zachytáva graf na obrázku 2.1.



Obr. 2.1: Graf, ktorý zachytáva vzťah medzi počtom tranzistorov na čipe vzhľadom k produktivite návrhárov (prevzaté z [1]).

Z grafu je vidieť, že medzi nárastom počtu tranzistorov a produktivitou vzniká istá medzera (nazývaná „Design Gap“), ktorá vyvíja tlak na zvyšovanie produktivity návrhárov.

2.1.1 Zvyšovanie produktivity návrhárov

Enormný nárast zložitosti navrhovaných obvodov teda vytvára tlak na zvyšovanie produktivity návrhárov. Istú paralelu je možné vidieť aj v prípade prechodu od popisu programov v jazyku symbolických inštrukcií až k objektovo orientovaným jazykom. Dnes sa programovanie v nízkoúrovňových jazykoch využíva najmä v kritických miestach aplikácií, kde je potrebná vysoká optimalizácia kódu a rýchlosť. Vysokoúrovňové jazyky poskytujú obrovskú mieru komfortu pre vývojárov, pretože prinášajú veľkú mieru abstrakcie od konkrétnej platformy. Čoraz viac je kladený dôraz na znovupoužiteľnosť kódu a zapúzdrenie jednotlivých modulov aplikácií.

Podobný vývoj je možné sledovať aj v prípade návrhu programovateľných obvodov. Zo začiatku sa využívalo manuálne kreslenie schém, poprípade jazyk ABEL-HDL. V roku 1981 začal vývoj nového HDL jazyka VHDL (VHSIC HDL), ktorý bol súčasťou armádneho programu Very-High Speed Integrated Circuit. Cieľom bolo vytvoriť jazyk, ktorý je nezávislý na cieľovej technológii a umožňuje návrh, simuláciu a verifikáciu číslicových obvodov. Tento jazyk bol v roku 1987 vydaný ako štandard IEEE. V roku 1984 sa objavuje jazyk Verilog, ktorý vyvíjala spoločnosť Gateway Design Automation. Tento jazyk sa stal štandardom IEEE v roku 1995.

Koncom deväťdesiatych rokov sa objavuje koncept IP jadier alebo Intelectual Property Core, ktorý umožňuje znovupoužiteľnosť celých navrhnutých blokov. IP jadro je zapuzdrený blok s pevne nakonfigurovanou logikou, ktorá implementuje samostatnú funkciu (FFT, prevod farebných modelov, ...). Takéto jadrá sa používajú pri FPGA aj pri ASIC návrhoch. Pre návrhára sú prezentované svojím vstupným a výstupným rozhraním, na základe ktorého je možné začleniť tieto bloky do návrhu. Toto urýchľuje návrh vďaka tomu, že sa do navrhovaného obvodu vkladajú už predpripravené bloky. Technika často umožňuje skrátiť dobu návrhu o celé mesiace, keďže sú dostupné celé knižnice takýchto blokov a je možné sa stretnúť so softwarovými, ale aj hardwarovými IP jadrami.

Problémom VHDL pre programátorov je často iný spôsob programovania. Ako názov napovedá, nejde priamo o programovací jazyk, ale o jazyk pre popis hardwaru, čo býva pre programátorov veľkým problémom. Väčšina programátorov však ovláda niektorý z imperatívnych programovacích jazykov (napr. C/C++). Preto boli v sedemdesiatych rokoch 20. storočia začaté výskumy, ktoré sa venovali možnosti prevodu takýchto jazykov na popis programovateľnej logiky. Takýto proces sa nazýva syntéza na systémovej úrovni a má na vstupe popis algoritmu vo vyššom programovacom jazyku a proces syntézy prevedie tento popis na zapojenie logických členov. V polovici deväťdesiatych rokov sa objavujú prvé komerčné nástroje, ktoré nemali väčší úspech. Patrí sem Behavioral Compiler (Synopsis) alebo Monet tool (Mentor Graphics). Problém týchto nástrojov bol predovšetkým v tom, že bol zvolený zlý vstupný jazyk, nástroj poskytoval zlú kvalitu výsledkov a problematickú verifikáciu výsledného obvodu. Avšak od roku 2000 sa objavujú úspešné nástroje, medzi ktoré patria: Catapult C Synthesis, Vivado HLS, C-to-Silicon, Handel-C alebo nedávno uvedený Stratus High-Level Synthesis.

Tieto nástroje prinášajú zvýšenie abstrakcie z popisu obvodov pomocou HDL jazykov, na popis prostredníctvom jazyka vyššej úrovne, ako je práve využitie jazykov C/C++ a jeho dialektov, ale sú využívané aj iné jazyky. Ako hlavnou výhodou je prezentovaná úspora času návrhu z mesiacov na týždne, jednoduchšia verifikácia, ladenie a čitateľnosť kódov.

2.2 Syntéza na systémovej úrovni

Syntéza na systémovej úrovni je definovaná ako proces, ktorý má na vstupe popis algoritmu vo vyššom programovacom jazyku (napríklad C, C++), knižnicu RTL komponentov cieľovej platformy a zoznam obmedzení. Na výstupe syntézy dostávame buď konfiguráciu pre FPGA alebo masku pre ASIC čip. Všeobecne pozostáva tento proces z nasledujúcich krokov [2]:

1. Kompilácia kódu a prevod do vhodnej formálnej reprezentácie (najčastejšie do formy podobnej Control Data Flow Graph, ktorý bude vysvetlený neskôr)
2. Alokácia hardwarových zdrojov, ktorá prebieha v procese plánovania a priradenia
3. Plánovanie operácií na jednotlivé hodinové takty a priradenie prostriedkov
4. Generovanie RTL schémy
5. Logická syntéza pomocou syntéznych nástrojov

2.2.1 Kompilácia a prevod na CDFG

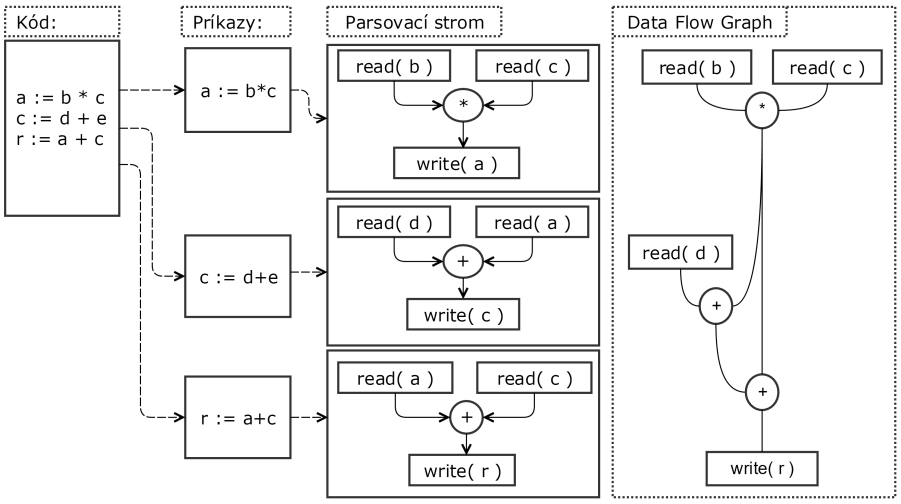
Proces HLS typicky začína prekladom funkčnej špecifikácie. Prvý krok tejto transformácie je prevod do formálnej reprezentácie. Klasicky sa oddeľujú dátové závislosti a tok riadenia.

Dátové závislosti môžu byť ľahko reprezentované grafom toku dát (DFG – Data Flow Graph), v ktorom každý uzol reprezentuje operácie alebo premenné. Hrany medzi uzlami predstavujú dátové závislosti. Formálne je definovaný [3]:

Definícia 2.1. DFG je orientovaný graf $G = (V, E)$, kde:

- $V = (v_1, v_2, \ldots, v_n)$ je množina uzlov
- $E \subseteq V \times V$ orientovaných hrán medzi uzlami, teda $e_i = (v_i, v_k)$

Zostrojenie tohto grafu je priamočiare a príklad je na obrázku 2.2. Prevod funguje tak, že pri kompilácií sa vytvorí syntaktický strom a z príkazov sa zostrojí odpovedajúci DFG.



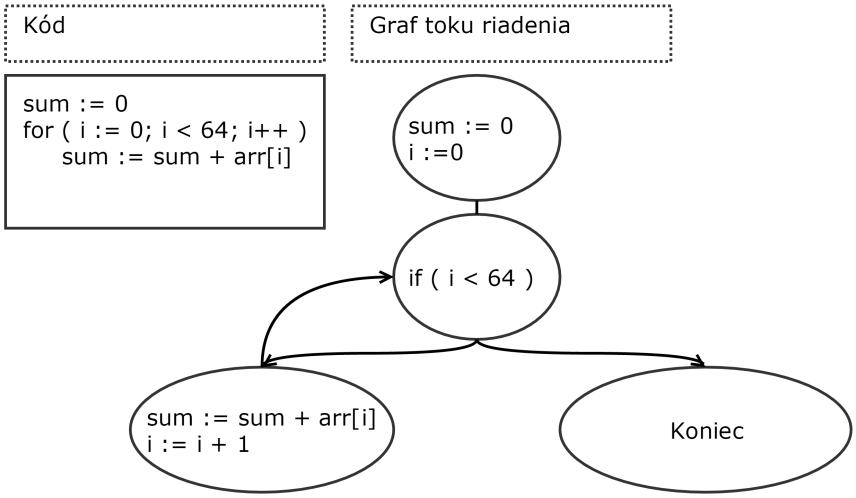
Obr. 2.2: Ukážka prevodu jednoduchého výrazu na DFG.

DFG síce zachytáva tok dát v algoritme, ale na druhej strane nezachytáva riadiace závislosti, kam patria podmienky alebo cykly. Modelovanie riadiacich závislostí je možné prostredníctvom grafu toku riadenia (CFG – Control Flow Graph), ktorý sa skladá z uzlov. Tieto uzly predstavujú atomickú postupnosť operácií. Hrany modelujú možnú postupnosť behu programu. Definícia je obdobná ako pri DFG:

Definícia 2.2. CFG je orientovaný graf $G = (V, E)$, kde:

- $V = (v_l, v_2, \ldots, v_n)$ je množina uzlov, ktoré predstavujú základne bloky kódu bez vetvenia
- $E \subseteq V \times V$ orientovaných hrán medzi uzlami, teda reprezentujú možný tok dát medzi blokmi

Každý takýto blok je tvorený postupnosťou operácií a vykonáva sa od prvej po poslednú operáciu. Tento prevod je takisto priamočiary a je zachytený na obrázku 2.3.



Obr. 2.3: Obrázok, ktorý ukazuje prevod jednoduchého výrazu na CFG.

Pre popis obvodu sa používa kombinácia CFG a DFG, ktorá sa nazýva Control Data Flow Graph (CDFG):

Definícia 2.3. CDFG je graf $G = (V, E)$, kde:

- $V = (v_1, v_2, \ldots, v_n)$ je množina uzlov, ktoré môžu predstavovať tok riadenia alebo zapúzdrovať DFG
- $E \subseteq V \times V$ orientovaných hrán medzi uzlami, hrany teda reprezentujú tok riadenia medzi uzlami

CDFG teda reprezentuje obvod spôsobom, v ktorom je reprezentované riadenie aj tok dát. Je možné ho prirovnať k intermediárnemu kódu, ktorý produkuje softwarový kompilátor. Avšak toto je iba teoretická definícia, prakticky vzniklo viacero spôsobov ako reprezentovať riadenie a dátový tok, medzi ktoré patria oddelené CDFG, de Jongov graf (definícia dostupná v publikácií [3]) alebo SSIM graf.

Transformácie

Takáto špecifikácia prostredníctvom CDFG však nie je vhodná pre priamy prevod na finálny popis obvodu na úrovni RTL. Avšak má výhodu v tom, že na tejto úrovni popisu algoritmu sa dajú vykonávať viaceré optimalizácie, ktoré sú svojou podstatou ekvivalentné optimalizáciám vykonávaným softwarovým kompilátorom [4]. Patria sem hlavne:

- eliminácia mŕtveho kódu
- propagácia konštánt
- eliminácia spoločných podvýrazov
- eliminácia nadbytočných operátorov
- vkladanie funkcií
- rozbalenie slučiek

Takisto sú často využívané transformácie, ktoré sú závislé na konkrétnej hardwarovej platforme. Často je takou transformáciou nahradenie násobenia a delenia číslom 2, ktoré sa nahradí jednoduchším bitovým posunom doľava resp. doprava.

2.2.2 Alokácia

Cieľom alokácie je minimalizovať počet hardwarových jednotiek potrebných na realizáciu výpočtu. Tieto jednotky sa vyberajú z knižnice, ktorá je dostupná pre danú hardwarovú platformu. Celý tento problém sa radí medzi NP-úplné problémy, takže sa využívajú heuristiky na vytvorenie sub-optimálnej alokácie. Tento krok sa delí na tri podúlohy:

Alokácia funkčných jednotiek často prebieha v procese plánovania. Výber jednotiek závisí na viacerých faktoroch, ako je plocha a oneskorenie, ktoré na sebe často závisia. Medzi takéto jednotky patria sčítačky s postupným prenosom vs. paralelné sčítačky.

Alokácia pamäťových jednotiek prebieha pri priraďovaní premenných do pamäťových prvkov. Patria sem rôzne registre, registrové polia, RAM pamäte, ROM pamäte a pod. Tento krok prebieha pri priraďovaní premenných do pamäťových prvkov.

Alokácia komunikačných ciest využitých na dátové prenosy. Je možné zvoliť medzi zdieľanou, multiplexovanou zbernicou alebo rôznymi špecifickými zbernicami.

Tieto jednotlivé kroky na sebe závisia a dosiahnutie optimalizovanej implementácie je vzhľadom k zložitosti jednotlivých častí obtiažne. Takisto je dôležité poznamenať, že jednotlivé kroky alokácie ovplyvňujú a sú ovplyvnené vybranými jednotkami. Každý HLS nástroj pristupuje k tomuto problému iným spôsobom. Jedným z jednoduchších spôsobov je obmedziť počet funkčných jednotiek a potom plánovať. Flexibilnejšou možnosťou je postupne cyklicky opakovať proces plánovania a alokácie.

2.2.3 Plánovanie

Základnou úlohou plánovania je vytvoriť plán výpočtu algoritmu, ktorý minimalizuje čas potrebný na vykonanie celého výpočtu, pričom na tento plán sú kladené obmedzenia. Medzi tieto obmedzenia patria napríklad plocha alebo celkový čas. Plánovanie je teda definované ako proces, ktorý priraďuje každej časovej jednotke, nazývanej kontrolný krok (C-step), jednotlivé operácie, ktoré sa v ňom vykonajú. Z hľadiska reprezentácie pomocou CDFG ide o rozdelenie tohoto grafu do podgrafov, ktoré sú priradené jednému kontrolnému kroku.

Počet operacií v podgrafe potom určuje počty a typy funkčných jednotiek, ktoré musia byť v tomto kroku k dispozícií (toto je dôvod prečo plánovanie ovplyvňuje alokáciu).

Plánovacie algoritmy

Keďže plánovanie je NP-úplný problém, vzniklo viacero algoritmov, ktorých cieľom je vytvoriť suboptimálny plán. Tieto algoritmy je možné rozdeliť do dvoch hlavných tried [4]. Prvú triedu tvoria **transformačné algoritmy**, ktoré začínajú so základným plánom, ktorý je zvyčajne maximálne paralelný alebo serializovaný a aplikujú sa transformácie za cieľom dosiahnutia iného plánu. **Iteračné algoritmy** vytvárajú plán pridávaním vždy jednej operácie, pokiaľ nie sú všetky naplánované.

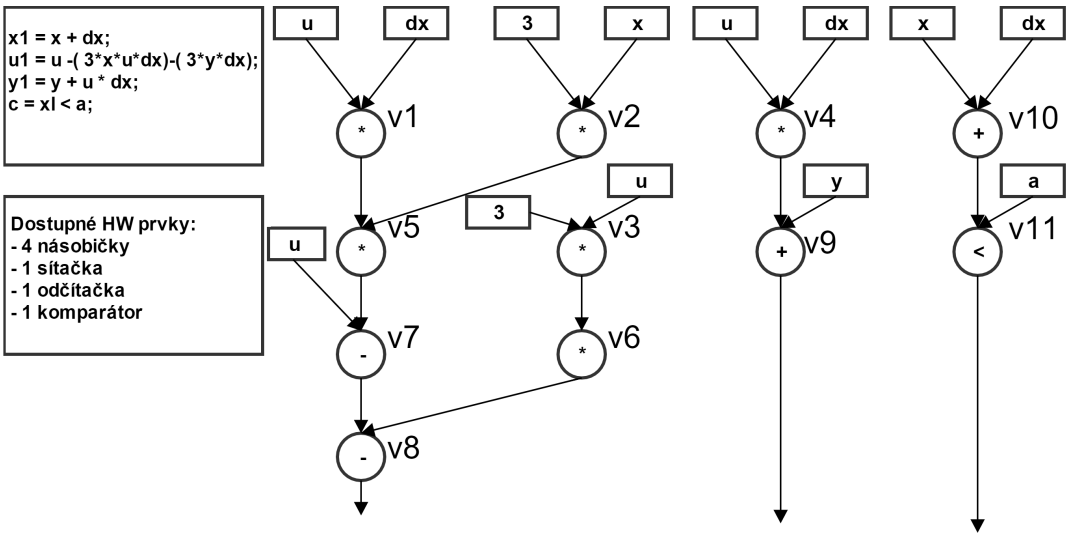
Transformačné algoritmy

Algoritmus EXPL je jeden z prvých plánovacích algoritmov, ktorý funguje na princípe lačného prehľadávania priestoru možných plánov. Patrí medzi transformačné algoritmy, takže prehľadávanie prebieha tak, že postupne vytvára jednotlivé plány a následne vyberie najlepší plán. Tento algoritmus nie je možné uplatniť na zložitejšie obvody, ale je možné ho upraviť na varianty, ktoré využívajú rôzne obmedzenia prehľadávania priestoru plánov.

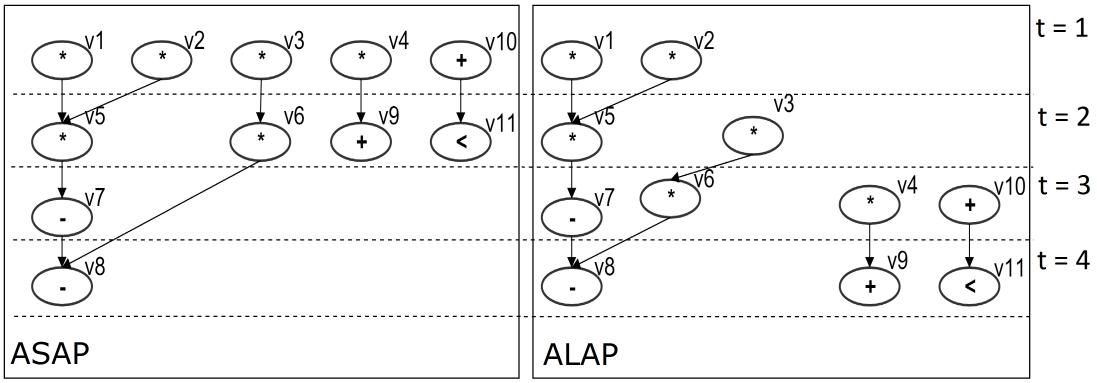
Inou možnosťou je vytvárať plán pomocou transformácií, ktoré sú volené na základe heuristík. Transformácie sú vyberané tak, aby viedli k splneniu daných obmedzení. Tento prístup využíva napríklad Yorktown Silicon Compiler a CAMAD design system.

Iteračné algoritmy

Inú triedu plánovacích algoritmov tvoria iteratívne. Medzi dva základne algoritmy patrí algoritmus **as soon as possible** (ASAP) a **as late as possible** (ALAP). ASAP postupuje tak, že naplánuje operácie hneď, keď to umožnia dátové závislosti. ALAP plánuje operácie čo najneskôr. Na príklade (obrázok 2.4), ktorý je často využívaný ako benchmarkový obvod, budú ukázané ich výsledky. Tieto algoritmy pracujú nad DFG grafom.



Obr. 2.4: Prevod výrazu na DFG, ktorý bude využitý pre algoritmy ASAP a ALAP (prevzaté z [5]).



Obr. 2.5: Obrázok, ktorý ukazuje prevod DFG, ktorý je na obrázku 2.4, na plán operácií podľa algoritmu ASAP a ALAP (prevzaté z [5]).

Na obrázku 2.5 je vidieť výsledok plánovacích algoritmov. Pri detailnejšom pohľade na výsledky oboch algoritmov sa dá usúdiť, že viacero operácií môže byť naplánovaných vo viacerých kontrolných krokoch. Ako príklad je možné uviesť uzol v_6 v algoritme ASAP, ktorý môže byť naplánovaný v časte $t = 2$ alebo $t = 3$ (ak sú dodržané všetky závislosti). Takto sa dá pre každý uzol vytvoriť interval, v ktorom môže byť naplánovaný. Tento rozdiel medzi maximálnym a minimálnym časom sa nazýva **mobilita operátoru**. Následne výsledky ALAP a ASAP využívajú algoritmy, ktoré vďaka nim označia hranice pre jednotlivé operátory. Patria sem algoritmy:

- Force-Directed Heuristic – snaží sa rovnomerne rozprestrieť operátory
- List-Based Scheduling – na vytvorenie plánu využíva zoznam operátorov, ktoré môžu byť naplánované a prioritnú funkciu, ktorá vyberá operátory

Samozrejme je tento pohľad na plánovanie dosť zjednodušený. Často sa vyskytujú prvky, ktoré vyžadujú viacero taktov na vytvorenie výstupu, poprípade existujú reťazené prvky, ktoré môžu začať výpočet v každom takte, ale výsledok je dostupný až po viacerých taktoch.

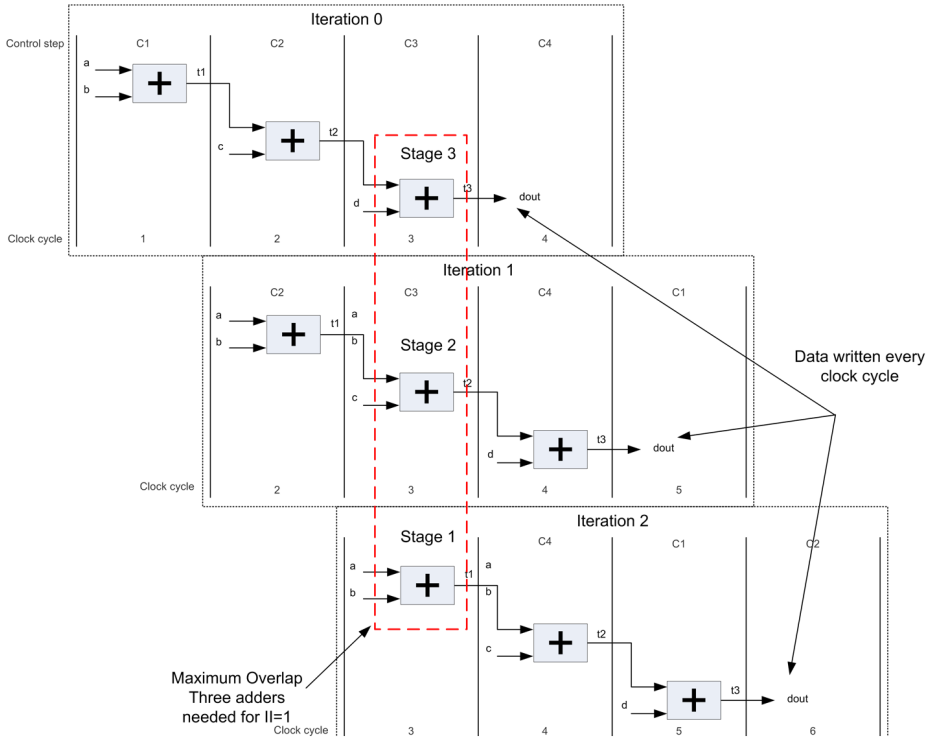
Plánovanie základných programových konštrukcií

Podkapitola bude zameraná na techniky, ktoré procesu HLS pridávajú možnosť modelovať paralelizmus. Je to hlavne prostredníctvom slučiek, ktoré poskytujú dizajnérom obrovské možnosti určovania vlastností výsledného obvodu. Táto podkapitola vychádza hlavne z [6].

Reťazenie – Pipelining

Pojem je veľmi podobný technike, ktorá sa využíva v RISC procesoroch, kde sa nazýva zreťazené spracovanie. Reťazená slučka (pipelined loop) umožňuje začať výpočet novej iterácie pred ukončením výpočtu aktuálnej a cieľom je zvýšenie priepustnosti. Princíp spočíva v rozdelení výpočtu na stupne, ktoré sú od seba oddelené pomocou registrov. Tieto registre sú rozmiestnené tak, aby kombinačná logika v každom takte vykonala určitú časť výpočtu. Toto umožňuje v nasledujúcom takte začať nový výpočet a po rozdelení na k stupňov je dosiahnuté až $k+1$ násobné zrýchlenie. Základný parameter, ktorý charakterizuje reťazenie sa nazýva inicializačný interval (Initiation Interval – II), ktorý určuje po koľkých cykloch začne výpočet novej iterácie. Na druhej strane istý čas trvá, kým dôjde k „napleneniu“ linky.

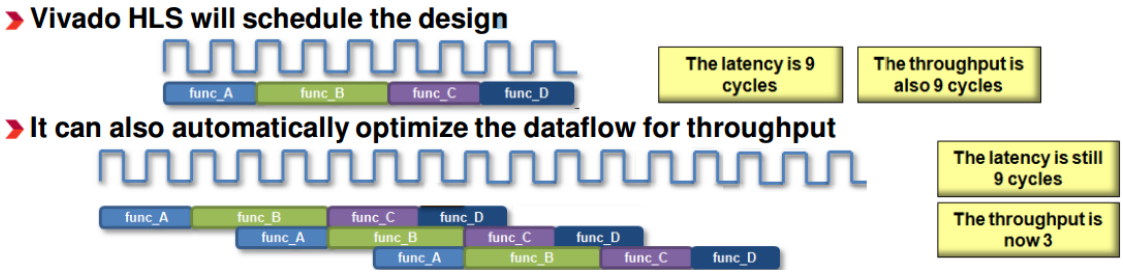
Tento čas sa nazýva latencia a pri zreťazení slučky sa ešte zvýši o čas, ktorý je spôsobený pridaním týchto registrov. Výhodou je oveľa väčšia priepustnosť. Obrázok 2.6 zachytáva princíp, kde je vidieť, že výsledok sa produkuje každý hodinový cyklus (II=1).



Obr. 2.6: Obrázok, ktorý znázorňuje reťazenie slučky. Ukázaná je naplánovaná slučka pomocou zreťazeného spracovania, kde každá iterácia slučky je prezentovaná samostatným „riadkom“ a c-step „stĺpcom“ (prevzaté z [6]).

Reťazenie funkcií – Dataflow

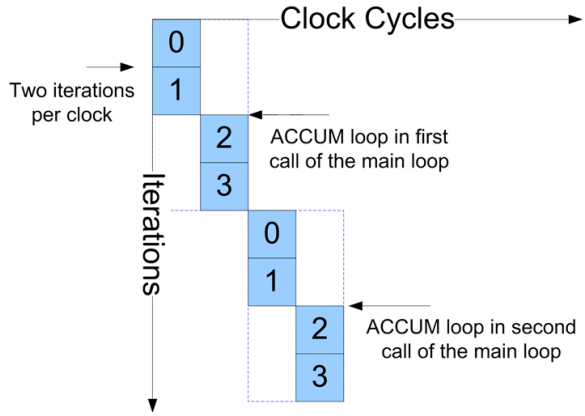
Ďalšou zaujímavou technikou zvyšovania priepustnosti je reťazenie na úrovni funkcií. Princíp je rovnaký ako pri reťazení slučiek, ale v tomto prípade je reťazený dizajn na úrovni funkcií. Je možné si túto techniku predstaviť ako jednotlivé stupne spracovania, kde tieto stupne (funkcie) si postupne predávajú dáta a vykonávajú nad nimi operácie. Samozrejme je možné využiť aj zreťazenie na úrovni slučiek, kde bude v každom stupni zreťazená slučka. Symbolický plán je znázornený na obrázku 2.7.



Obr. 2.7: Obrázok, ktorý znázorňuje reťazenie na úrovni funkcií (prevzaté z [7]).

Rozbalenie slučky – Loop Unrolling

Táto technika umožňuje niekoľkokrát skopírovať slučku a tým zrýchliť výpočet. Ako je vidieť na obrázku 2.8, paralelizmus rastie s počtom paralelných iterácii. Každou kópiou slučky sa znásobuje plocha a počet jednotiek, preto často sa využíva čiastočné rozbalenie slučky (partial unrolling), ktoré je určené faktorom rozbalenia, tj. koľkokrát má byť rozbalená.



Obr. 2.8: Obrázok, ktorý znázorňuje rozbalenie slučky, kde je možné vidieť že iterácie 0,1 sú spočítané v prvom cykle a iterácie 2,3 v druhom. Čiže je dosiahnuté dvojnásobné zrýchlenie za cenu dvojnásobného počtu zdrojov (prevzané z [6]).

Spájanie slučiek – Loop Merging

Táto technika má za úlohu minimalizovať plochu. Toto je možné, ak sú dve slučky s podobnými vlastnosťami. Základný princíp je v tom, že sa využije iba jeden cyklus, ktorý obsahuje spojené telá všetkých funkcií. Výhodou je odstránenie réžie, ktorú vytvára inicializácia cyklu. Príklad na túto techniku je na algoritme 2.1 a výsledok na algoritme 2.2.

```
1  void topFunc ( /* vstup */ ) {
2    L1: for ( i=3;i>=0;i--) {
3      loop1Func( );
4    }
5
6    L2: for( i=3;i>=0;i--){
7      L3: for( j=3;j>=0;j--) {
8        loop23Func( );
9      }
10   }
11
12   L4: for( i=3;i>=0;i--) {
13     loop4Func( );
14   }
15 }
```

Algoritmus 2.1: Príklad pre ukázanie spájania slučiek. Každá slučka vyžaduje vlastné riadenie cyklu. Z algoritmu je vidieť, že slučky majú podobné vlastnosti.

```
1  void topFunc ( /* vstup */ ) {
2    L1234: for ( l=9,l>=0;l--) {
3      if ( cond1 ) {
4        loop1Func( );
5      }
6
7
8      loop23Func( );
9
10
11     if ( cond4 ) {
12       loop4Func( );
13     }
14   }
15 }
```

Algoritmus 2.2: Manuálne spojenie slučiek. Podmienkami `cond1` a `cond4` sa špecifikujú iterácie, v ktorých prebehne spracovanie tela pôvodnej slučky.

Loop flattening

Pri riešení praktických úloh sa často vyskytuje úprava matíc (spracovanie obrazu, ...). Vnorené slučky vytvárajú réžiu, ktorá je spojená s výpočtom jej indexu. Táto réžia má vplyv na výslednú latenciu obvodu. Technika spočíva v odstránení tejto réžie tým, že telo vnútornej slučky je duplikované tak, akoby sa počítalo viac riadkov matice v jednom cykle.

Zhodnotenie plánovania

Ako bolo ukázané v tejto podkapitole, plánovanie predstavuje ťažiskovú úlohu procesu syntézy. Najskôr boli rozobrané základne princípy pre plánovanie a nakoniec bola pozornosť venovaná technikám, ktoré umožňujú ovplyvňovať vytvorený plán. Dizajnér má možnosť vytvárať plány s ohľadom na rôzne kritéria ako latencia, priepustnosť, plocha na čipe alebo použité zdroje.

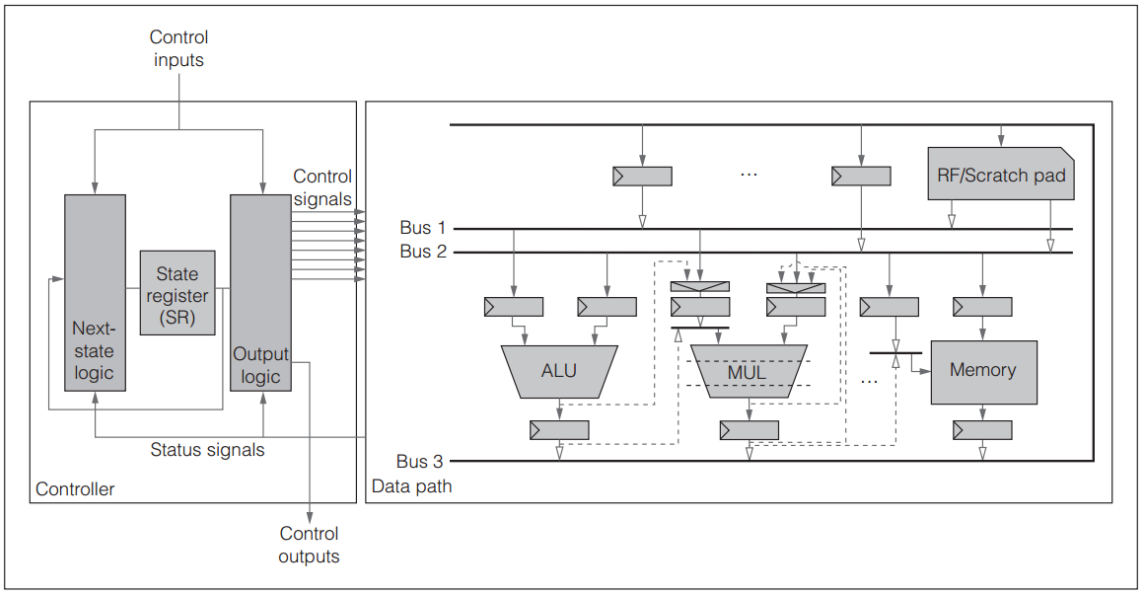
2.2.4 Priradenie

V kroku priradenia zdrojov je za cieľ namapovať premenné a operácie už naplánovaného CDFG grafu do funkčných jednotiek, pamäťových jednotiek a prepojovacích prvkov. Plánovanie aj priradenie má teda veľký vplyv na vlastnosti výsledného obvodu. Priradenie sa delí na tri typy:

- **Priradenie funkčných jednotiek** slúži na výber vhodných jednotiek, ktoré implementujú jednotlivé funkcie. Tieto jednotky sa nachádzajú v tzv. knižnici komponentov, ktorá obsahuje aj ich detailnú implementáciu. Knižnica komponentov cieľovej platformy môže obsahovať viacero realizácií tej istej funkčnej jednotky, ale s inými vlastnosťami (požadované zdroje, latencia, pracovná frekvencia). Proces priradenia musí s ohľadom na požiadavky vybrať vhodnú realizáciu.
- **Priradenie pamäťových jednotiek** zaručuje pre každú premennú, ktorá je použitá pri výpočte, naviazanie na pamäťovú jednotku. Často je možné počas jedného výpočtu vybrať premenné tak, že sa doby ich života neprekrývajú a môžu byť mapované do tej istej pamäťovej jednotky.
- **Priradenie prepojovacích prvkov** ovplyvňuje priradenie pamäťových a funkčných jednotiek, pretože často závisí na možnostiach prepojenia medzi nimi. Dôvodom je, že sú potrebné prenosy z funkčnej jednotky do inej a na to sú využité prepojovacie jednotky (zbernice, multiplexory). Samotný proces HLS odhaduje oneskorenie prepojení aby bolo možné lepšie optimalizovať výsledný dizajn.

2.2.5 Generovanie výslednej RTL schémy

V poslednom kroku procesu HLS sa podľa výsledkov procesov alokácie, plánovania a priradenia generuje výsledná schéma obvodu. Tento výstup môže mať rôzne podoby, ako napríklad VHDL, Verilog, IP Core a iné. Výsledný obvod má zvyčajne tvar radiča a dátových ciest. Takúto obecnú schému je možné vidieť na obrázku 2.9. Dátové cesty pozostávajú z množiny pamäťových jednotiek, množiny funkčných jednotiek a prepojeniami medzi nimi. Radič je tvorený konečným stavovým automatom, ktorý riadi tok dát v dátových cestách nastavovaním riadiacich signálov. Radič pozostáva zo stavového registra, logiky pre ďalší stav a výstupnej logiky [2].



Obr. 2.9: Na obrázku je zobrazená finálna realizácia obvodu, ktorá pozostáva z radiča a dátových ciest (prevzaté z [2]).

2.3 Vivado High-Level Synthesis

V tejto kapitole je pozornosť venovaná nástroju pre proces HLS, ktorý bude využitý pri praktickej časti. Týmto nástrojom je Vivado HLS od spoločnosti Xilinx, ktorý slúži na transformáciu popisu obvodu v jazyku C, C++ alebo SystemC na špecifikáciu na úrovni RTL implementácie. Keďže ide o komerčný produkt, nie je možné zistiť kompletný popis procesu, ale spoločnosť Xilinx uvoľnila viacero dokumentov, ktoré popisujú detaily. Kapitola vychádza zo zdrojov [7] a [8]. Nasleduje popis jednotlivých krokov, ktoré sa veľmi približujú procesu HLS, tak ako bol popísaný v predchádzajúcej podkapitole. Vivado vykonáva dva druhy syntézy:

- Algoritmická syntéza – prevádza funkcie a syntetizuje funkčnú špecifikáciu na RTL schému.
- Syntéza rozhrania – prevádza parametre funkcií na RTL porty so špecifickým časovaním, šírkou a pod. Rozhranie top-level funkcie (funkcia, ktorá reprezentuje vstupný a výstupný bod komponentu) sa stáva rozhraním pre finálny komponent.

Proces prevodu v prípade Vivado HLS pozostáva z týchto krokov:

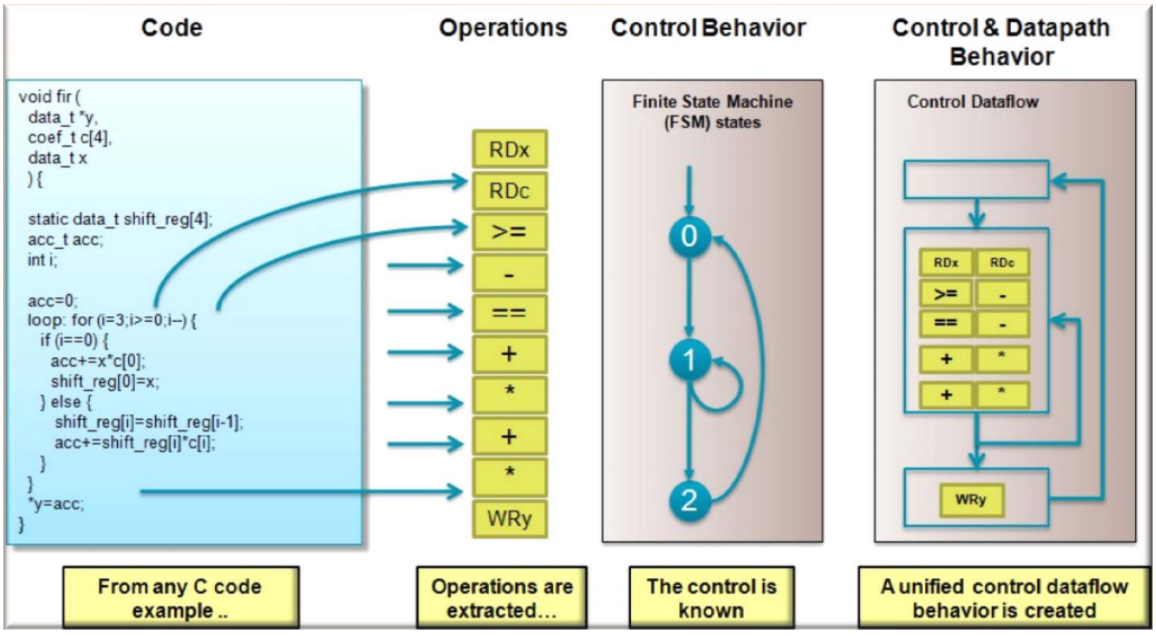
- Prevod do vhodnej reprezentácie obvodu – extrakcia toku dát a riadiacej logiky
- Plánovanie – mapovanie operácií do hodinových taktov
- Priradenie – mapovanie hardwarových prvkov, ktoré realizujú jednotlivé operácie
- Optimalizácie

2.3.1 Prevod do vhodnej reprezentácie obvodu

Táto fáza ma za úlohu vytvoriť vhodnú reprezentáciu popisovaného obvodu, ktorá bude zachytávať všetky podstatné vlastnosti implementácie, a bude dostatočne nezávislá na použitom HDL jazyku. V prípade Vivado HLS tento krok prebieha vo viacerých fázach:

- Extrakcia operácií (Operation extraction)
- Extrakcia riadiacej logiky (Control Behavior)
- Vytvorenie jednotného popisu algoritmu v ktorom je zachytený tok riadenia aj tok dát (Control and Datapath Behavior)

Tento proces je veľmi pekne zachytený na obrázku 2.10, kde môžeme vidieť, že Vivado HLS využíva prevod do určitej formy CDFG. Keďže Vivado HLS spĺňa štandardy prekladačov GCC alebo G++, je možné túto reprezentáciu algoritmu chápať ako intermediárny kód v prípade softwarového kompilátoru. V týchto krokoch je vykonané množstvo optimalizácií, ktoré vykonávajú aj ostatné syntézne nástroje alebo kompilátory.



Obr. 2.10: Obrázok, na ktorom môžeme vidieť prevod kódu jednoduchého FIR filtru na internú reprezentáciu (prevzaté z [7]).

2.3.2 Plánovanie

V tejto fáze sa určuje pre každý kontrolný krok množina operácií, ktoré sa v ňom vykonajú. Základ tvoria tieto podmienky:

- Dĺžka hodinového cyklu
- Tok riadenia a tok dát
- Čas, ktorý vyžaduje operácia na vykonanie (tento čas závisí na cieľovej platforme)
- Užívateľské obmedzenia

Samozrejme proces syntézy automaticky určí, ktoré operácie môžu byť hotové v jednom hodinovom takte, a ktoré vyžadujú viac hodinových taktov. Takto je dosiahnutá abstrakcia od konkrétnej HW platformy.

2.3.3 Priradenie

V tejto fáze sa priraďuje konkrétny prvok knižnice cieľovej platformy každej plánovanej operácii. Aby bolo implementované optimálne priradenie, HLS navyše využíva informácie o cieľovej platforme. Nástroj Vivado HLS kladie dôraz na možnosti výberu vhodného použitia zdrojov [7]. Nasledujúci príklad na plánovanie ukazuje plán pre rýchlejšiu a pomalšiu frekvenciu hodín:

Obr. 2.11: Obrázok obsahuje dva plány operácií pre pomalšiu a rýchlejšiu frekvenciu hodín (prevzaté z [7]).

Teraz je možné urobiť nasledujúce rozhodnutia ohľadom výberu jednotiek, ktoré budú realizovať výpočet:

- Zdielať – v prípade plánu na ľavej strane je potrebné použiť 2 násobičky a je možné aj rozhodnutie, či sa použije oddelená sčítačka a odčítačka alebo zdieľaná jednotka pre sčítanie a odčítanie.
- Nezdieľať – v prípade plánu na pravej strane môžeme použiť buď dve násobičky alebo jednu zdieľanú.

Z predchádzajúceho príkladu je vidieť vplyv priradenia na výslednú realizáciu obvodu, a ako následne môže ovplyvniť plánovanie. Teda ak hodiny majú dlhší hodinový takt alebo je použitá rýchlejšia FPGA technológia, môže byť viac operácií dokončených v rámci jedného taktu. Naopak v prípade, že hodinový takt je kratší alebo je použitá pomalšia FPGA, Vivado automaticky naplánuje operácie na viac hodinových cyklov, pričom niektoré môžu byť realizované cez ich väčší počet.

2.3.4 Optimalizácie Vivado HLS

Nástroj poskytuje možnosti pre optimalizáciu výsledného dizajnu. Podľa dokumentu [8] sú optimalizácie zamerané na priepustnosť, latenciu, plochu a logické optimalizácie. Veľký počet optimalizácií umožňuje obrovskú variabilitu pri návrhu dizajnov.

2.4 Programovací model Vivado HLS

V tejto podkapitole nasleduje oboznámenie so špecifikami programovania pre správny proces HLS. Najskôr je stručne opísaný nástroj Vivado HLS, potom je pozornosť venovaná hlavným detailom programovania a nakoniec prehľad špeciálnych akcelerovaných knižníc.

2.4.1 Vivado Design Suite

Vivado Design Suite je balík softwarových aplikácií od spoločnosti Xilinx pre návrh, syntézu, optimalizáciu a verifikáciu číslicových obvodov. Je nástupcom Xilinx ISE, oproti ktorému pribudla hlavne syntéza na systémovej úrovni a interný simulátor. Konkrétne pozostáva z Vivado High-Level Synthesis, Vivado IP Integrator, System Generator for DSP a Xilinx SDK.

Z pohľadu tejto diplomovej práce je najzaujímavejšie práve vývojové prostredie Vivado HLS, ktoré poskytuje rozhranie pre riadenie procesu syntézy. Umožňuje správu projektov, tvorbu a úpravu zdrojových kódov, syntézu a jej riadenie, funkčnú verifikáciu a pod.

2.4.2 Hlavné zdroje informácií

Medzi hlavné zdroje informácií patrí dokument **UG902**[8], ktorý obsahuje kompletný popis metodiky HLS. Obsahom sú aj komentované príklady, ktoré ukazujú základy zápisu algoritmov vo forme vhodnej pre syntézu. Druhým zdrojom sú príklady, ktoré je možné spustiť priamo vo Vivade, kde sa nachádzajú rôzne vzory často používaných dizajnov. Ďalším miestom, kde hľadať riešenie už detailnejších a závažnejších problémov, je oficiálne fórum pre Vivado HLS[1].

2.4.3 Vivado HLS

Prostredie Vivado HLS je postavené na vývojovom prostredí Eclipse a umožňuje podobnú správu projektov a zdrojových kódov. Keďže ide o komerčné vývojové prostredie, množstvo funkcií je upravených, napríklad je možné mať otvorený iba jeden projekt. Pre vytváranie nových projektov je využitý jednoduchý sprievodca, ktorým sa nadefinujú základné vlastnosti dizajnu. V sprievodcovi je nadefinovaná top level funkcia, následne perióda hodín a cieľová platforma. Po tomto kroku je už možné vytvárať zdrojové kódy a súbory. Priamo v tomto prostredí prebieha ladenie dizajnu na úrovni vysokoúrovňového kódu, je možné spustiť syntézu alebo kosimuláciu. Podstatná je aj možnosť prehliadania výsledkov syntézy, kde sa dajú prehliadať plány operácií, využitie zdrojov alebo pridelenie jednotiek na jednotlivé c-step.

2.4.4 Štruktúra projektu vo Vivade

Projekt v Vivado HLS obsahuje viacero typov súborov, ktoré sú potrebné pre samotný proces HLS:

- C, C++ alebo SystemC súbory – obsahujú funkcie, ktoré majú byť syntetizované. V prípade jednoduchého dizajnu môže obsahovať jeden súbor alebo v prípade zložitejšieho, obsahuje mnoho súborov a funkcií (súbory `.c`, `.cpp`, `.h`).

[1] `https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/bd-p/hls`

- C testbench súbory – základ pre ladenie dizajnu a funkčnú verifikáciu obvodu po procese syntézy (súbory `.c`, `.cpp`, `.h`, ale sú priradené do zložky Test Bench).
- Direktívy (Directives) – riadia proces syntézy a ovplyvňujú výslednú formu obvodu (súbory s príponou `.tcl`).

Ďalšou časťou projektu sú tzv. riešenia, ktoré reprezentujú jednotlivé dizajny. Úlohou riešení je možnosť jednoducho vytvárať rôzne dizajny (rôzne platformy, rôzne časovania, rozbalenia slučiek, ...).

Po procese syntézy stačí vyexportovať dizajn a medzi výstupy patria:

- SystemC model – RTL výstup HLS je určený iba pre RTL simuláciu.
- VHDL a Verilog – Syntetizovateľný kód, ktorý môže byť integrovaný do projektu a použitý na generovanie bitstreamu pre programovanie FPGA alebo Zynq zariadení.
- IP jadro pre Vivado, System Generator alebo XPS – vhodné pre priame vloženie do Vivado IP Integrator, XPS projektu alebo Xilinx System Generator.

V nasledujúcich kapitolách je pod pojmom kód v jazyku C myslený kód zapísaný v C, C++ alebo SystemC.

2.4.5 Riadenie procesu syntézy

Pre riadenie procesu syntézy je možné využiť jednu z nasledujúcich možností:

- pragma direktívy, ktoré sa priamo vkladajú do zdrojového kódu
- pomocou súboru TCL (Tool Command Language), ktorý obsahuje direktívy previazané s dizajnom pomocou návestí alebo názvov premenných

Výhodou využitia pragma direktív je umiestnenie priamo v zdrojovom kóde a teda sú rýchlejšie prístupné. Avšak nevýhoda je v tom, že ak je vytvorených viacero riešení, je nutné ich prispôsobiť pre konkrétne riešenie. Naproti tomu umiestnenie direktív do súboru TCL umožňuje mať pre každé riešenie vlastný TCL súbor. Na nasledujúcom príklade zreťazenia slučky je ukázané zapísanie pomocou pragma direktívy (algoritmus 2.3) a pomocou direktívy v TCL súbore (algoritmus 2.4).

```
1  ...
2  for( int i=0; i<128; i++ )
3  {
4  #pragma HLS pipeline II=1
5    dataOut[i] = dataIn[i] + 6;
6  }
7  ...
```

Algoritmus 2.3: Ukážka zápisu direktívy pre zreťazenie funkcie pomocou pragma direktívy.

```
1  ...
2  LOOPI: for( int i=0; i<128; i++ )
3  {
4    dataOut[i] = dataIn[i] + 6;
5  }
6  ...
7
8  /* obsah suboru .tcl */
9  set_directive_pipeline
10                 loop_pipeline/LOOPI
```

Algoritmus 2.4: Ukážka zapísania príkazu pre zreťazenie funkcie prostredníctvom .tcl súboru.

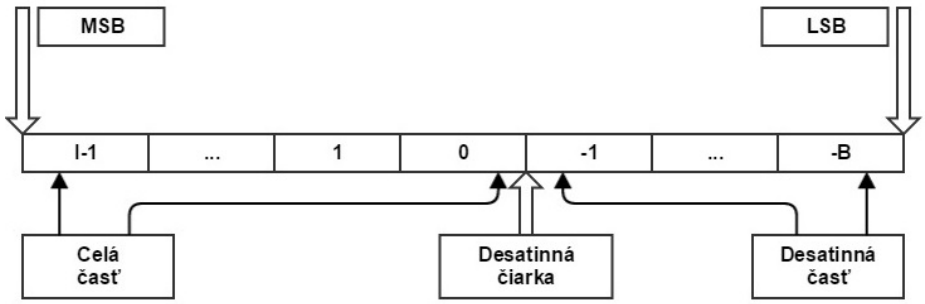
2.4.6 Dátové typy

Jednou zo základných vlastností Vivado HLS je široká podpora dátových typov. Patria sem základne znamienkové aj bezznamienkové dátové typy (char, short int, int, long int, float, double), štruktúry, polia a samozrejme objekty. Veľkou výhodou je možnosť určenia dátových typov s vlastnou bitovou šírkou, čo umožňuje tvorbu optimalizovaného dizajnu pre potreby aplikácie. Je možné takto navrhnúť vlastné celočíselné dátové typy, ale aj typy s pevnou desatinnou čiarkou. V prípade C++ je potrebné využiť hlavičkový súbor <ap_int.h> pre celočíselné dátové typy a pre dátové typy s pevnou desatinou čiarkou <ap_fixed.h> . Samozrejme sa dá využiť aj typ float alebo double, ale operácie nad týmito typmi vyžadujú väčšie množstvo zdrojov. Na nasledujúcom príklade 2.5 sú ukázané možnosti definovania dátových typov s vlastnou bitovou šírkou.

```
1 #include <ap_int.h>
2
3 typedef ap_int< 18 > myInt18;   // celociselny znamienkovy datovy typ
4 typedef ap_uint< 18 > myUInt18; // celociselny beznamienkovy datovy typ
```

Algoritmus 2.5: Príklad ako definovať vlastný celočíselný dátový typ, ktorý má bitovú šítku 18 bitov

Typ s pevnou rádovou čiarkou je reprezentovaný ako typ, ktorý má určitý počet bitov pre celočíselnú časť a určitý počet bitov pre desatinnú časť čísla. Sémantika jednotlivých častí je zobrazená a vysvetlená na obrázku 2.12.



Obr. 2.12: Obrázok, na ktorom je zobrazená reprezentácia čísla v pevnej rádovej čiarke. Hodnota I vyjadruje počet bitov celej časti a B vyjadruje počet bitov pre reprezentovanie desatinnej časti.

V prípade pevnej rádovej čiarky sú možnosti konfigurácie dátového typu omnoho väčšie. Šablóna má pre znamienkový typ tvar `ap_fixed< W, I, Q, O, N >` a pre beznamienkový dátový typ `ap_ufixed< W, I, Q, O, N >` . Vysvetlenie jednotlivých parametrov šablóny:

- W – šírka celého dátového typu
- I – počet bitov pre reprezentáciu celej časti čísla (Platí: $W = I + B$)
- Q – mód kvantizácie čísla, keď je požadované uloženie čísla, ktoré vyžaduje väčšiu presnosť ako poskytuje užívateľský dátový typ
- O – správanie pri pretečení čísla
- N – definuje počet saturačných bitov pri pretečení v móde `wrap`

2.4.7 Funkcie

Keďže v prípade HLS neexistuje vstupný bod programu ako v prípade klasického SW programu, určuje sa tzv. top-level funkcia. Táto funkcia sa následne stáva rozhraním komponentu v RTL dizajne. Podobne všetky volané funkcie sú reprezentované ako komponenty a pre každú funkciu je vytvorený po procese syntézy vlastný report a výstupný RTL súbor (VHDL, Verilog, SystemC).

Volané funkcie môžu byť voliteľne **inlinované** (function inlining), čo znamená zlúčenie implementácie volanej funkcie s miestami, kde je funkcia volaná. Tento prístup môže zvýšiť možnosti optimalizácií, ale je potrebný väčší počet logických členov. V prípade, že je funkcia inlinovaná, nie je generovaný žiaden výsledný report.

Ďalším dôležitým krokom je správny návrh funkcií a jej argumentov. Vhodným návrhom argumentov funkcie sa dá ovplyvniť celkový dizajn. Pri vhodnom využití dátových typov s vlastnou bitovou šírkou je možné dosiahnuť optimálne využitie výpočetných jednotiek. Veľmi výstižný príklad je zobrazený na algoritmoch 2.6 a 2.7, kde v prípade algoritmu 2.6 sa vysyntetizuje dizajn, ktorý bude využívať dva 32 bitové vstupy, 32 bitovú násobičku a jej výstup bude orezaný na 24 bitov. V druhom prípade sa využije už 24 bitová násobička so vstupmi o veľkosti 12 bitov.

```
1  #include <ap_cint.h>
2
3
4  typedef int24 doutT;
5
6  doutT mult32(int x, int y)
7  {
8    int tmp = (x * y);
9    return tmp;
10 }
```

Algoritmus 2.6: Ukážka funkcie, ktorá sa syntetizuje s použitím 32 bitovej násobičky.

```
1  #include <ap_cint.h>
2
3  typedef int12 dinT;
4  typedef int24 doutT;
5
6  dout_t mult24(dinT x, doutT y)
7  {
8    int tmp = (x * y);
9    return tmp;
10 }
```

Algoritmus 2.7: Ukážka funkcie, ktorá sa syntetizuje s použitím 24 bitovej násobičky.

2.4.8 Reťazenie slučiek

Reťazenie slučiek bolo predstavené v kapitole 2.2.3. Teraz na praktickom príklade algoritmu 2.8 bude ukázané, aké dopady má reťazenie slučiek na výsledný dizajn. Základným pravidlom je, že ak je reťazená slučka, tak všetky vnútorné slučky musia byť úplne rozbalené:

- **Zreťazená slučka LOOP_J** spôsobí iba jednu kópiu LOOP_J v dizajne, HLS využije vonkajšiu slučku LOOP_I na prísun nových dát. Bude naplánované iba jedno násobenie a jeden prístup do pamäte.
- **Zreťazená slučka LOOP_I** spôsobí úplne rozbalenie tela slučky LOOP_J, tj. musí byť naplánovaných 20 násobení a 20 prístupov do pamäte.
- Ak bude zreťazená top-level fukcia musia byť rozbalené obidve slučky: plánovanie 400 násobení a 400 prístupov do pamäte.

```
dout_t loopPipeline(din_t A[N])
{
  static dout_t acc;
  LOOP_I:for( int i = 0; i < 20; i++ ){
    LOOP_J: for( int j = 0; j < 20; j++ ){
     acc += A[i] * j;
     }
  }
  return acc;
}
```

Algoritmus 2.8: Funkcia, na ktorej sú prezentované vlastnosti reťazenia.

Dizajnér môže zvoliť vhodný typ reťazenia, avšak vždy sa dostane k trade-off riešeniu[1] medzi latenciou a počtom zdrojov. Reťazením **LOOP_J** bude latencia asi 400 cyklov (20x20) a bude použitých menej ako 100 LUT a registrov. Reťazením **LOOP_I** bude latencia cca 20 cyklov, ale bude použitých niekoľko stoviek LUT a registrov. Reťazením funkcie bude latencia približne 10 cyklov, ale počet LUT a registrov bude niekoľko tisíc.

2.4.9 Perfektná a semiperfektná slučka

So slučkami sa ešte viažu dva pojmy. Prvým je tzv. **perfektná slučka**. Takáto slučka má telo (tj. výpočet) iba v najvnútornejšej slučke, neexistuje žiadna dátová závislosť medzi príkazmi a hranice slučky sú konštantné. Druhým je **semi-perfektná** slučka, ktorá sa líši v tom, že vonkajšia slučka môže mať ako ukončujúcu podmienku premennú. Semi-perfektná slučka vyžaduje dodatočné hodinové cykly pre prológ a epilóg. Niekedy sa dajú prepísať takéto slučky na perfektné, čo umožní väčšie možnosti pre reťazenie.

2.4.10 Paralelizmus slučiek

Primárnou snahou Vivado HLS je znižovanie latencie tak, že sa snaží plánovať logiku a výpočet čo najskôr to dovolia závislosti. Ak nie je možné spojiť slučky, je tu možnosť využiť paralelizmus slučiek, kde Vivado HLS naplánuje dve nezávislé slučky aby bežali paralelne. Toto sa deje automaticky bez súčinnosti dizajnéra.

2.4.11 Polia

Polia sú zvyčajne implementované ako pamäte RAM, ROM alebo FIFO. Polia, ktoré sú argumenty top-level funkcie sú syntetizované ako RTL porty, ktoré pristupujú k pamäti pomocou zbernice. Polia použité vo funkciách sú synetizované do blokových RAM, LUT-RAM alebo prípadne registrov. Tak ako v prípade slučiek, tak aj v prípade polí je možné využiť množstvo optimalizácií a úprav, ktoré vedú na rôzne vlastnosti výsledného dizajnu. Najväčšiu pozornosť vyžaduje sledovanie a analyzovanie týchto kľúčových bodov:

- Prístup k poliam často vytvára tzv. „úzke hrdlo“ aplikácie, keďže prístup do pamäte vždy tvá istú dobu (nezanedbateľnú).
- Inicializácia polí – ak nie je dôkladná, spôsobuje neočakávane chovanie.
- Rozhodnutie či nemôže byť pole implementované ako ROM pamäť.

[1] Trade-off riešenie je situácia, v ktorej sa musí vybrať vhodný kompromis, kde nie je možné dosiahnuť všetky požadované vlastnosti.

Práca s pamäťou sa často stáva tzv. úzkym hrdlom aplikácie (tzv. Bottleneck) a ukážka je na jednoduchom príklade, ktorý je zapísaný v algoritme 2.9, kde sa pole syntetizuje ako RAM. V tomto prípade sú potrebné tri súčasné prístupy do pamäte. Ak je využitá jedno-portová (jeden zápis a čítanie v jednom takte) alebo dvoj-portová pamäť, stáva sa táto funkcia „úzkym hrdlom“ aplikácie. Ďalšou nevýhodou je, že sa nedá zreťaziť s inicializačným intervalom II=1, keďže proces syntézy upozorní na prekročenie počtu prístupov do pamäte v jednom takte.

```
dout_t array_mem_bottleneck(din_t mem[N])
{
  dout_t sum=0;
  SUM_LOOP:for(int i=2;i<N;++i)
    sum += mem[i] + mem[i-1] + mem[i-2];
  return sum;
}
```

Algoritmus 2.9: Ukážka bottlenecku v prípade prístupu k poľu.

Tento kód sa dá prepísať na optimalizovaný prístup do pamäte, ktorý je už možné reťaziť a optimalizovať. Je to vďaka tomu, že sú manuálne prednačítané dáta z RAM pamäte a v tomto prípade stačí aj jedno-portová RAM:

```
dout_t array_mem_perform(din_t mem[N])
{
  din_t tmp0, tmp1, tmp2;
  dout_t sum=0;
  tmp0 = mem[0];
  tmp1 = mem[1];
  SUM_LOOP:for (int i = 2; i < N; i++)
  {
    tmp2 = mem[i];
    sum += tmp2 + tmp1 + tmp0;
    tmp0 = tmp1;
    tmp1 = tmp2;
  }
return sum;
}
```

Algoritmus 2.10: Ukážka odstránenia problému s prístupom do pamäte.

FIFO pole je často používané v mnohých aplikáciach a Vivado HLS umožňuje syntézu polí ako FIFO. Prístup do takéhoto poľa musí byť prísne v sekvenčnom poradí a začínať od nuly. Ak sa pristupuje na rôznych miestach funkcie, musí byť takisto zaručené poradie. Implementácia FIFO poľa je jednoduchá a zápis sa nachádza na algoritme 2.11:

```
void array_FIFO (dout_t d_o[4], din_t d_i[4])
{
  For_Loop: for( int i=0; i < 4; i++ )
  {
    d_o[i] = d_i[i];
  }

}
```

Algoritmus 2.11: Ukážka FIFO poľa, ktoré jednoducho skopíruje vstup na výstup.

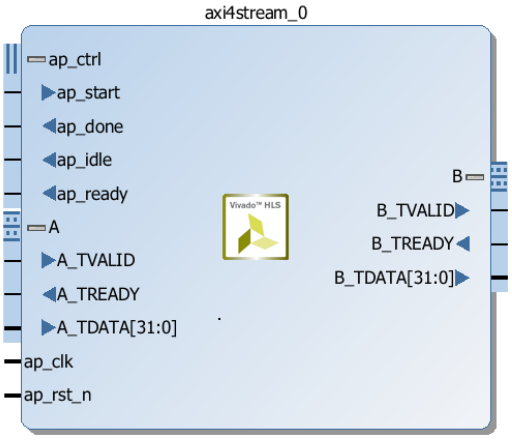
2.4.12 Syntéza rozhrania

Vivado poskytuje dve možnosti ako špecifikovať rozhranie komponentu. Najpohodlnejším je syntéza rozhrania priamo z argumentov top-level funkcie, kde jednotlivé argumenty sú prevedené na porty a dajú sa určiť rôzne protokoly prístupu – AXI a Block-Level I/O protokoly. Pre AXI je možnosť ďalej špecifikovať AXI4-Stream, AXI4-Lite alebo AXI4 master rozhranie. Druhou možnosťou je definícia prostredníctvom TCL príkazu.

Block–Level I/O protokoly sú `ap_ctrl_none`, `ap_hs`, `ap_ctrl_chain`, `ap_stable` `ap_hs` (ap_ack, ap_vld, ap_ovld), `ap_memory`, `bram`, `ap_fifo` `ap_bus`. Z ich názvu je často zrejmá ich funkčnosť, poprípade ich popis sa nachádza v dokumente UG902.

AXI4–Stream rozhranie je ukázané na nasledujúcom príklade 2.4.12, kde proces syntézy správne vygeneroval všetky potrebné signály pre tento protokol. Ďalej existuje možnosť pridať vlastné signály na tento port pomocou šablón. Tieto signály sa nazývajú postranné (side-channels) a dajú sa využiť ako doplnkové riadiace signály.

```
1  typedef int din_t;
2  typedef int dout_t;
3
4  void axi4stream( din_t A[128],
5                   dout_t B[128] )
6  {
7  #pragma HLS INTERFACE axis port=A
8  #pragma HLS INTERFACE axis port=B
9
10   for( int i = 0; i < 128; i++ )
11   {
12     B[i] = A[i] << 2;
13   }
14 }
```



Algoritmus 2.12: Ukážka syntézy funkcie s využitím AXI4-Stream.

AXI4-Lite rozhranie umožňuje, aby komponent bol riadený CPU alebo mikrokontrolérom, ďalej poskytuje zoskupenie viacerých portov na jedno rozhranie AXI4-Lite a prenos menších dátových prenosov. Vivado HLS navyše umožňuje exportovanie driveru v jazyku C, ktorý je možné skompilovať pre procesor. Zápis pomocou pragma deklarácií sa nachádza v algoritme 2.13, kde je vidieť aj zoskupovanie portov na viacero AXI rozhraní.

```
1  void example( int A[ 128 ], int B[ 128 ], int C[ 128 ] )
2  {
3  #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
4  #pragma HLS INTERFACE s_axilite port=A bundle=BUS_A
5  #pragma HLS INTERFACE s_axilite port=B bundle=BUS_B
6  #pragma HLS INTERFACE s_axilite port=C bundle=BUS_A
7
8    for( int i = 0; i < 128; i++ )
9      C[i] = ( A[i] << 2 ) + B[i];
10 }
```

Algoritmus 2.13: Ukážka zápisu funkcie s využitím AXI4-Lite. Kontrolné rozhranie a porty premenných **a** a **c** budú prenášané cez spoločné rozhranie. Port premennej **b** bude priradená na vlastné rozhranie.

AXI4 Master rozhranie sa používa na poliach alebo ukazovateľoch a je možné ho využiť v dvoch režimoch. Prvým sú samostatné dátové prenosy, druhým sú dávkové prenosy dát s použitím funkcie memcpy. Ukážka sa nachádza na nasledujúcom algoritme 2.14.

```
1  #define COLS 1280
2  #define ROWS  720
3  void memcpyAndInc(volatile char *a)
4  {
5  #pragma HLS INTERFACE m_axi depth=1280 port=a
6  #pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS
7    // port bude priradeny AXI4 Master rozhraniu
8    char buff[COLS];
9    for( int row = 0; row < ROWS; row++ )
10   {
11     // memcpy realizuje davkovy pristup do pamate
12     memcpy(buff,(const char*)(a + row*cols),cols*sizeof(char));
13     for( int col = 0; col < COLS; col++ )
14       buff[i] = buff[i] + 1;
15     memcpy((char*)(a + row*cols),(const char*)buff,cols*sizeof(char));
16   }
17 }
```

Algoritmus 2.14: Ukážka zápisu funkcie s využitím AXI4 Master.

2.4.13 Nesyntetizovateľné konštrukcie

Nie všetky konštrukcie, ktoré sa dajú zapísať v programovacom jazyku C, je možné syntetizovať. Medzi nesyntetizovateľné konštrukcie patria:

- **Systémové volania** sú nesyntetizovateľné, keďže pri ich volaní je očakávaná istá interakcia s jadrom OS. Vivado HLS ignoruje najpoužívanejšie systémové volania, ale aj tak je dobrou praktikou odstrániť takéto volania pred procesom HLS. Tak ako pri programovaní iných aplikácií, aj tu je vhodné využiť konštrukciu #ifdef, ktorú poskytuje preprocesor jazyka C/C++. Proces HLS navyše automaticky generuje symbol „`__SYNTHESIS__`“ pri volaní preprocesoru. Jednoducho je teda možné počas ladenia kódu volať systémové volania. Prínos je hlavne pri testovaní, kde je umožnená práca so súbormi, dynamickou alokáciou pamäte a podobne.
- Nie je dovolené využívať **dynamicky alokovanú pamäť** a teda všetky takéto konštrukcie musia byť zmenené pred procesom syntézy na pevne stanovené limity.
- **Ukazovatele** sú síce podporované a Vivado HLS umožňuje ukazovatele na polia, aj na jednotlivé prvky skalárnych polí, ale neumožňuje ukazovateľ na ukazovateľ.
- **Pretypovanie** nie je povolené pri zložených dátových typoch, ale iba pri natívnych typoch.
- **Rekurzia** nie je syntetizovateľná. Dokonca aj funkcie, ktoré by mohli potencionálne viesť na nekonečnú rekurziu sú nesyntetizovateľné.
- **STL** knižnica (Standard Template Library) je nesyntetizovateľná, pretože obsahuje dátové typy, ktoré využívajú rekurziu a dynamickú alokáciu pamäte.
- Niektoré matematické funkcie nie sú syntetizovateľné, ako príklad sa dá uviesť funkcia mocniny.

2.4.14 Test Bench

Jedným zo spôsobov ako zvyšuje Vivado rýchlosť vývoja je pomocou Test Bench, v ktorom ide o ladenie C implementácie algoritmu, ktorý má byť syntetizovaný. V tomto prístupe je porovnávaný známy výsledok s výsledkom implementovaného algoritmu pre syntézu. Je to z dôvodu, že ladenie na úrovni jazyka C je oveľa rýchlejší ako simulácia na úrovni RTL. Ďalšou vlastnosťou Vivada je použitie tohoto Test Bench súboru aj pre simuláciu na úrovni RTL. Ná základe Test Bench funkcie a vysyntetizovaného dizajnu sa zostaví testovací obvod.

Odporúčané je mať implementované dve top-level funkcie – jedna pre syntézu a jedna pre Test Bench (klasická funkcia main(...)). V prípade Test Bench funkcie je dôležité, aby jej návratová hodnota zodpovedala zaužívanému štandardu, kde 0 znamená test skončil v poriadku a nenulová hodnota značí ukončenie testu s chybou. Dôvodom je, že Vivado podľa návratovej hodnoty funkcie určuje výsledok testu. Ladenie C/C++ implementácie prebieha obdobne ako v iných vývojových prostrediach, kde je možné krokovať program, zadávať breakpointy a sledovať hodnoty premenných. Pre ladenie využíva Vivado HLS ladiaci program GDB (GNU Project debugger).

2.4.15 Kosimulácia

Po procese syntézy je možné overiť funkčnosť dizajnu pomocou kosimulácie. Tento typ simulácie používa C testbench súbor pre generovanie vstupných stimulov. Pre simuláciu sa dá zvoliť jeden zo štyroch simulátorov (ModelSim, Vivado Simulator, ISE alebo Riviera). Na základe návratovej hodnoty test bench súboru je rozhodnutý aj výsledok kosimulácie. Ako poslednou možnosťou je vytvoriť súbory pre simuláciu, ktorú je možné následne prehliadať v spomenutých simulátoroch.

2.4.16 Akcelerované knižnice

Vivado HLS obsahuje viacero predpripravených knižníc, ktoré sú priamo určené pre určitý typ aplikácií. Jednou z najčastejšie používaných je určite <hls_stream.h>, ktorá umožňuje modelovať takýto dátový prenos. Medzi najzákladnejšie patrí aj <hls_math.h>, ktorá obsahuje podporované matematické funkcie pre syntézu (napr. log, exp, ...). Ďalšou knižnicou je <hls_video.h>, ktorá umožňuje základnú prácu s obrazom (zmena merítka, základné filtre, ...). Poskytnuté je aj prepojenie s knižnicou OpenCV prostredníctvom knižnice <hls_opencv.h>. Ďalej je obsiahnutá podpora pre Rýchlu Fourierovú transformáciu alebo lineárnu algebru. Všetky knižnice sú písané s dôrazom na znovupoužitieľnosť a to pomocou šablón.

2.5 Zynq-7000 All Programmable SoC

Podkapitola je zameraná na popis platformy Xilinx Zynq, na ktorej bude v druhej časti práce realizovaná aplikácia s využitím Vivado HLS. Nasleduje popis architektúry a hlavných vlastností tejto platformy. Text podkapitoly vychádza hlavne zo zdrojov [9] a [10].

Charakteristickým znakom platformy Zynq je spojenie dvojjadrového ARM Cortex-A9 procesoru s FPGA technológiou. Programovateľná logika FPGA je postavená na siedmej sérií FPGA spoločnosti Xilinx. Síce takéto platformy sú tu už dlho, avšak nemajú rovnaké vlastnosti ako Zynq, ktorého hlavný prínos spočíva v tom, že umožňuje beh štandardných operačných systémov, ako napríklad distribúcie Linuxu.

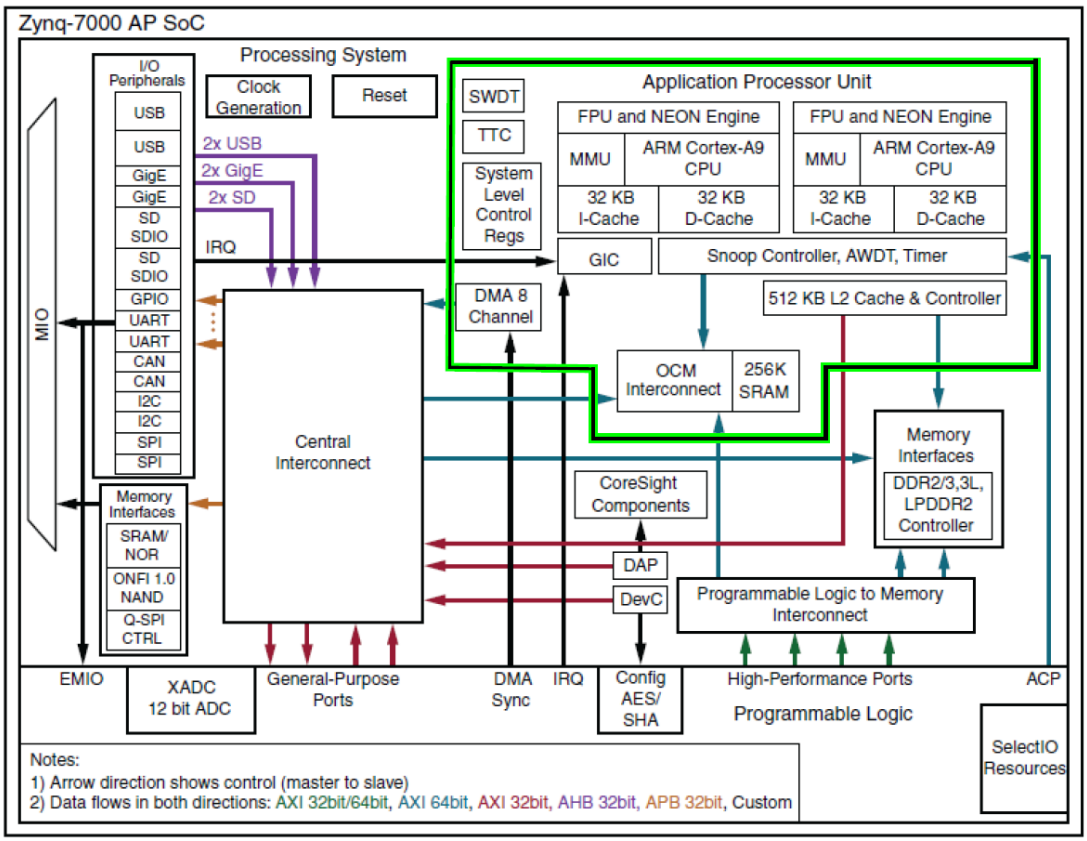
2.5.1 Architektúra Zynq

Jadro platformy Zynq tvorí ARM Cortex-A9 MPCore, ktorý môže byť taktovaný maximálne na 1 GHz. Tento procesor má dve jadrá a prostredníctvom ARM AMBA AXI prepojení komunikuje s pamäťami a periférnymi zariadeniami. Tieto časti spolu tvoria tzv. Processing system (PS). ZYNQ na jednom čipe okrem PS obsahuje aj programovateľnú logiku (Programmable Logic– PL), ktorá je prepojená s PS pomocou spomenutého ARM AMBA AXI rozhrania.

Týchto AXI prepojení je viacero a umožňujú efektívne prepojenia PS a PL. Konkrétne ide o dve 32 bitové AXI Master rozhrania, dve 32 bitové AXI Slave rozhrania, štyri 64 bitové konfigurovateľné AXI Slave rozhrania a jedno 64 bitové AXI ACP rozranie. Vďaka tomu je možné, že viacero hardwarových akcelerátorov môže mať vysokorýchlostný prístup do hlavných pamätí. Ak je potrebné mať koherentný prístup do pamätí cache, je nutné využiť 64 bitové ACP (Accelerator Coherency Port) prepojenie, ktoré je pripojené priamo na kontrolnú jednotku vyrovnávacích pamätí procesoru (Snoop Control Unit).

2.5.2 Processing system

Na obrázku 2.13 je možné vidieť architektúru Zynq a zvýraznenú časť PS, ktorá sa nazýva Application Processing Unit (APU). PS teda neobsahuje iba procesor ARM, ale aj radu na neho naviazaných akcelerátorov, pamätí, generátor hodín a rozhraní. Tieto časti spolu vytvárajú APU.



Obr. 2.13: Architektúra platformy Zynq (prevzaté z [9]).

APU obsahuje dve procesorové jadrá ARM, pričom každé má vyhradené vlastné výpočtové jednotky:

- NEON^{TM} Media Processing Engine (MPE)
- výpočtovú jednotku pre pohyblivú rádovú čiarku (FPU)
- jednotku správy pamäte (MMU), ktorá prekladá virtuálne adresy na fyzické a naopak
- L1 Cache s veľkosťou 32KB (oddelená pre inštrukcie a dáta, každá 32KB)

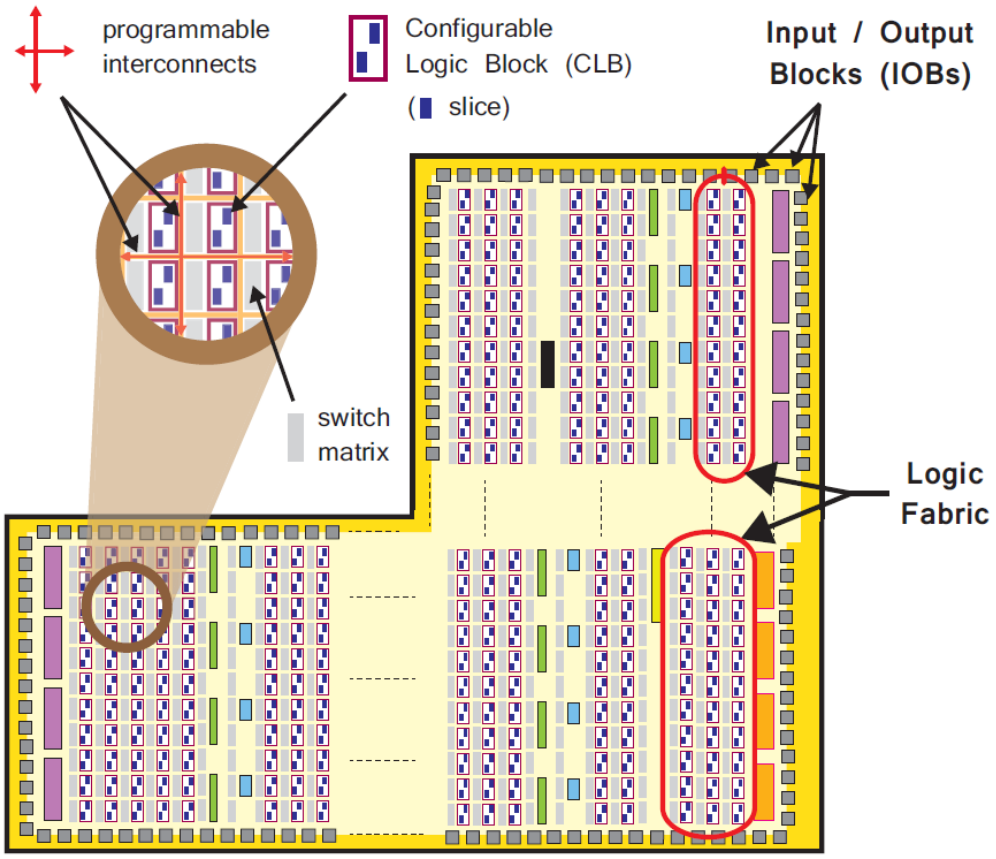
APU ďalej obsahuje 512KB L2 cache, ktorá je zdieľaná a pamäť na čipe (OCM– On Chip Memory) o veľkosti 256KB, ďalej už spomenutý Snoop Control Unit, ktorý vytvára spojenie medzi procesorovými jadrami a týmito pamäťami. Ďalej riadi transakcie medzi PS a PL pomocou už spomenutého Accelerator Coherency Port (ACP).

Týmto je tvorené jadro APU, ale aby bola zaistená flexibilita tejto platformy, obsahuje radu periférnych zberníc, ktoré zabezpečujú prepojenie s externými komponentmi. Toto je dosiahnuté pomocou Multiplexed Input/Output (MIO), ktoré prostredníctvom 54 konfigurovateľných pinov umožňuje flexibilne mapovať piny na periférie podľa potreby. Ďalej je možné využiť Extended MIO (EMIO), ktorý je zdieľaný s I/O zdrojmi PL. EMIO môže byť teda použité ako rozšírenie konektivity, poprípade na komunikáciu s IP blokom v PL, keďže tieto piny sú zdieľané. Zynq poskytuje nasledovné periférne zbernice a rozhrania:

- 2x SPI – sériové periférne rozhranie
- 2x I^2C – I^2C zbernica
- 2x CAN – podporuje CAN 2.0A a CAN 2.0B štandardy.
- 2x UART
- 4x GPIO port – 4 porty GPIO, každý obsahuje 32 pinov
- 2x SDIO – rozhranie pre pripojenie SD karty
- 2x USB – USB2.0, podrpouje host, device alebo OTG mód
- 2x GigE – ethernet 10Mbps, 100Mbps a 1Gbps mód

2.5.3 Programmable Logic

Programovateľná logika je založená na Artix-7 alebo Kintex-7 FPGA. PS a PL môžu byť úzko alebo voľne prepojené (úzko– nižšia latencia, voľne– vyššia latencia) vďaka spomenutým rozhraniam a ďalších signálov (podľa dokumentácie až 3000 spojov). To umožňuje dizajnérom integrovať hardvérové akcelerátory a ďalšie funkcie v PL, ktoré sú prístupné procesoru a tiež prístup týchto akcelerátorov do pamäte v PS. Fyzická štruktúra programovateľnej logiky je zobrazená na obrázku 2.14 a je vidieť, že obsahuje viacero typov komponentov. Počej jednotlivých jednotiek v PL sa líši pre každý model Zynq a taktiež je jedným z faktorov, ktoré je potrebné zohľadňovať pri výbere platformy.



Obr. 2.14: Na obrázku sa nachádza štruktúra PL, je vidieť stĺpcové usporiadanie prvkov(prevzaté z [9]).

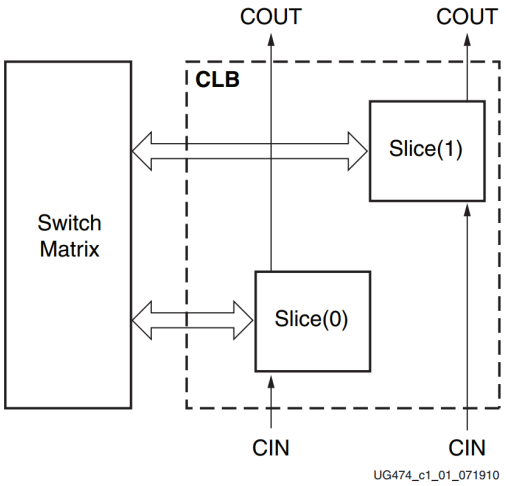
Ako je možné vidieť na predchádzajúcom obrázku 2.14, štruktúra PL je usporiadaná do matice a skladá sa z:

- Matice konfigurovateľných logických blokov (CLB)
- Block RAM
- DSP48E1 Slice – predpripravené bloky pre veľmi rýchle aritmetické operácie
- Prepojovacej matice – je tvorená horizontálnymi a vertikálnymi vodičmi, na ktoré sú pripojené jednotlivé prvky
- Konfigurovateľných vstupno/výstupných blokov (I/O Blocks – IOBs)
- Ďalej manažment hodín, nízko energetické sériové transceivery (závisí od modelu), integrované rozhranie pre PCI Express a mnoho ďalších komponentov

CLB

CLB alebo konfigurovateľné logické bloky sú zložené z dvoch slices, kde každý slice je zložený zo štyroch šesť-vstupových Look Up tabuliek (LUT) a ôsmich pamäťových elementov. Tieto

dva slices nemajú priame prepojenia medzi sebou, ale každý z nich je pripojený na prepojovaciu maticu. Každý takýto slice v stĺpci má vlastný carry reťazec pre realizáciu rýchlych aritmetických operácií. Zapojenie a štruktúra CLB je zobrazená na obrázku 2.15.



Obr. 2.15: Štruktúra jedného CLB (prevzaté z [10]).

Tieto LUT môžu byť využité ako:

- Šesť vstupové look-up tabuľky s jedným výstupom
- Dve päť vstupové look-up tabuľky
- Distribuovaná pamäť – RAM alebo ROM
- Posuvný register
- Zložité multiplexory

Block RAM

PL obsahuje Block RAM pamäte, pričom každý blok je dvoj-portový s veľkosťou 36Kb a maximálnou šírkou 72 bitov. Sú integrované priamo do hradlového poľa v stĺpcovom usporiadaní a sú vhodné pre ukladanie dát na PL. Ide o synchrónnu pamäť, takže každý prístup do pamäte je riadený hodinovým signálom. Túto pamäť je možné využiť ako RAM, ROM alebo FIFO pamäť.

Každá BRAM môže byť konfigurovaná ako jedna 36Kb RAM, alebo ako dve nezávislé 18Kb pamäte. Základná šírka slova je 18 bitov, takže každý blok tejto pamäte môže uchovávať 2048 položiek. Zaujímavá je aj vlastnosť nastavenia veľkosti prvku, vďaka čomu je možné zväčšovať alebo zmenšovať veľkosť prvkov. Nič nebráni tomu aby pamäť obsahovala 4096 9bitových položiek alebo 512 72bitových.

DSP48E1 slice

Tento slice má jednoduchú štruktúru, obsahuje predsčítačku (anglicky „pre-adder“), násobičku (anglicky „multiplier“) a voliteľnú operáciu (sčítanie, odčítanie a logická jednotka). Jeden z najcharakteristickejších použití, je implementácia symetrickej formy FIR filtra,

ktoré sa bežne vyskytujú pri digitálnom spracovaní signálov.Celý filter môže byť vytvorený kaskádou takýchto slicov spoločne. Sú špeciálne navrhnuté pre vysoký výkon a vysokú efektivitu. Ďalej sú využívané pre konštrukciu operácií v plávajúcej rádovej čiarke alebo zložitejších operácií.

2.5.4 Rozhranie PS–PL

Ako bolo spomenuté na začiatku tejto kapitoly, PS je pomocou prepojenia, ktoré spĺňa štandard ARM AMBA AXI. ARM AMBA protokol je otvorený štandard, ktorý špecifikuje prepojenie na čipe pre spojenie a manažment funkčných blokov na System-on-Chip (SoC). Tento štandard je tvorený protokolmi, medzi ktoré patria:

- CHI – Coherent Hub Interface
- ACE – AXI Coherency Extensions
- AXI – Advanced eXtensible Interface
- AHB – Advanced High-Performance Bus
- APB – Advanced Peripheral Bus
- ATB – Advanced Trace Bus

Samotné AXI obsahuje tri protokoly. Prvým je AXI4, ktorý je vhodný na komunikáciu mapovanú do pamäte, kde po poslaní adresy môže nasledovať dávka až 256 dátových slov („data beats“). Druhou možnosťou je AXI4-Lite a ako vyplýva z názvu, ide o zjednodušenú linku, kde po adresovaní nasleduje jedno dátové slovo. Posledným je AXI4-Stream, ktorý realizuje vysokorýchlostné streamované dáta, podporuje dávkový prenos bez obmedzenia veľkosti a je bez adresovacieho mechanizmu.

Na platforme Zynq je hlavným prepojením medzi PS a PL prostredníctvom 9 AXI rozhraní a každé sa skladá z viacerých kanálov. Týchto 9 kanálov vytvára dedikované spojenie medzi PS a PL. Rozhranie je point-to-point spojenie pre prenos dát, adries a synchronizačných (hand-shaking) signálov medzi master a slave zariadením. Toto prepojenie je tvorené:

- 2x 32-bit AXI master rozhranie (master je PS)
- 2x 32-bit AXI slave rozhranie (master je PL)
- 4x 64-bit konfigurovateľné AXI slave rozhranie
- 1x 64-bit AXI ACP (Accelerator Coherency Port) rozhranie

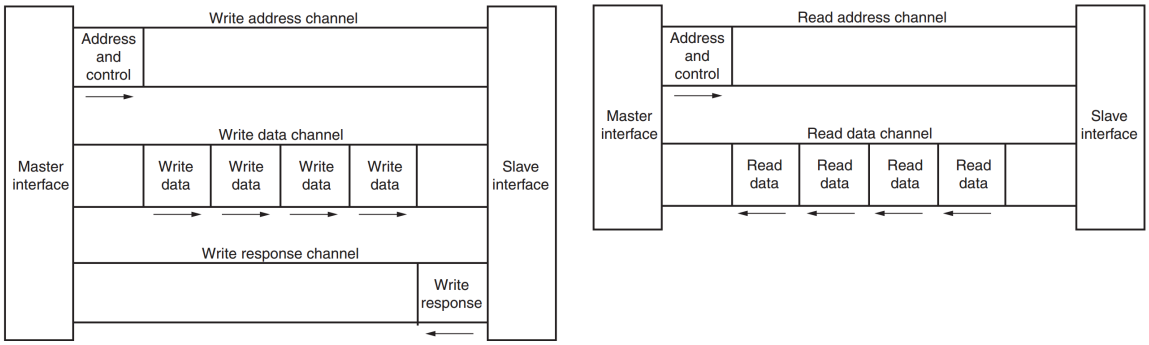
2.5.5 ARM AMBA AXI

AXI je teda súčasťou ARM AMBA protokolu, kde prvá verzia AXI bola súčasťou AMBA 3.0, uvedeného v roku 2003. AMBA 4.0 bola uvedená v roku 2010 a zahŕňa druhú významnú verziu AXI4, ktorá zahŕňa tri typy rozhraní:

- **AXI4** – Vysoko-rýchlostné prenosy, ktoré sú mapované do pamäte
- **AXI4-Lite** – Nízko-rýchlostné prenosy mapované do pamäte (kontrolné registre a pod.)
- **AXI4-Stream** – Vysoko-rýchlostné posielanie dát

Dáta môžu byť posielané v oboch smeroch od master aj slave zariadenia súčasne a veľkosť prenášaných dát sa môže líšiť. Ako bolo spomenuté v predchádzajúcom texte, AXI4 umožňuje v dávke prenos až 256 dátových slov. AXI4-Lite umožňuje počas jednej transakcie prenos dát jedným smerom. Protokol prenosu je jednoduchý a je zobrazený na schéme, ktorá je na obrázku 2.16. Architektúra AXI4 a AXI4-Lite rozhranie pozostáva z 5 kanálov:

- Kanál pre čítanie adresy (Read Address Channel)
- Kanál pre zápis adresy (Write Address Channel)
- Kanál pre čítanie dát (Read Data Channel)
- Kanál pre zápis dát (Write Data Channel)
- Kanál pre zápis odpovede (Write Response Channel)



Obr. 2.16: Popis protokolu AXI4 a AXI4-Lite, vpravo sa nachádza čítanie a vľavo zápis (prevzaté [11]).

AXI4-Stream protokol je vhodný pre aplikácie, ktoré sú zamerané na dátovo centrické a data-flow algoritmické problémy, kde adresovanie nie je dôležité. Dáta sú posielané od Master k Slave zariadeniu bez adresy. Každý AXI4-Stream poskytuje jeden jednosmerný kanál, ktorý je riadený handshakingom. Popis signálov sa nachádza v tabuľke 2.1.

Signál	**Zdroj**	**Popis**
ACLK	Zdroj hodín	Hodinový signál
ARESETn	Reset	Resetovací signál
TDATA	Master	Dáta
TVALID	Master	Označenie platných dát
TREADY	Slave	Signál určuje pripravenosť na prijatie dát
TSTRB	Master	Označuje, či TDATA obsahuje dáta alebo pozíciu
TKEEP	Master	Označuje neplatné dáta
TLAST	Master	Označuje koniec paketu
TID	Master	Identifikácia streamu
TDEST	Master	Využitie pre smerovanie
TUSER	Master	Postranný kanál

Tabuľka 2.1: Tabuľka obsahuje popis signálov pre rozhranie AXI4-Stream.

AXI4-Stream Video protokol je rozšírenie AXI4-Stream protokolu. Je určený pre prenos videa medzi PS a PL. Rozširuje AXI4-Stream protokol tým, že pridáva synchronizačné signály riadiace prenos jednotlivých snímok. AXI4-Stream Video je kompatibilný s AXI4-Stream komponentmi a využíva dve jednoduché úpravy protokolu:

- signál Start Of Frame (SOF) – určuje prvý pixel novej snímky
- signál End Of Line (EOL) – určuje posledný pixel každého riadku

Tieto signály sú potrebné na identifikáciu miesta pixelu v AXI4 streame, keďže nemá žiadnu adresáciu. Popis rozhrania sa nachádza v tabuľke 2.2.

Funkcia	**Šírka**	**Názov signálu**	**Názov**
Video data	až 1028 bitov	TDATA	DATA
Valid	1	TVALID	VALID
Ready	1	TREADY	READY
Start Of Frame	1	TUSER	SOF
End Of Line	1	TLAST	EOL

Tabuľka 2.2: Popis signálov pre rozhranie AXI4-Stream Video.

Veľkou výhodou je aj možnosť využitia DMA pre riadenie prenosu dát z pamäte, ktorý umožňuje vysoko-rýchlostný prístup medzi pamäťou a AXI4-Stream FPGA komponentmi. Poskytuje cyklický prístup k frame bufferom, ktorých môže byť až 16. Existuje možnosť „parkovania“ (anglicky park on a frame) na snímke a posielať ju opakovane. Dôležité je aj to, že každý kanál môže využívať inú rýchlosť prenosu snímok a pixelov.

Kapitola 3

Snímky s vysokým dynamickým rozsahom

Kapitola obsahuje úvod do problematiky HDRI (High-dynamic-range imaging). Najskôr je pozornosť venovaná základom snímania prostredníctvom digitálneho fotoaparátu. Následne je objasnený pojem HDR spolu so súvisiacimi pojmami. Ďalšia časť kapitoly popisuje spôsoby záznamu a zobrazovania HDR. Posledná časť kapitoly sa venuje metódam mapovania tónov, ktoré sú ťažiskové pre túto diplomovú prácu.

3.1 Problematika HDR

Dnes je najbežnejším fotoaparátom digitálny fotoaparát. Zjednodušene povedané, digitálny fotoaparát je analógovo-digitálny prevodník, ktorý prevádza svetlo dopadajúce na senzor na digitálnu informáciu. Tento proces sa nazýva expozícia a je definovaná ako vystavenie senzoru fotoaparátu svetlu zo snímanej scény. Pre riadenie expozície sa využívajú nasledovné prostriedky:

- Clona – otvor v objektíve fotoaparátu, ktorého veľkosť sa dá meniť. Zmenou veľkosti clony sa reguluje množstvo svetla, ktoré dopadá na svetelný senzor.
- Doba expozície – doba, počas ktorej dopadá svetlo na šošovku fotoaparátu. Zmenou tejto doby sa ovplyvní aj výsledný vzhľad scény.
- Citlivosť snímača – označovaná ako ISO, riadi citlivosť snímača na dopadajúce svetlo.

Dnešné bežné fotoaparáty zaznamenávajú obraz ako 8 alebo 12 bitovú informáciu na jeden farebný kanál pixelu a hodnoty intenzít snímajú lineárne. Linearita znamená, že ak je snímač osvetlený dvojnásobne silnejším svetlom, prejaví sa to ako zdvojnásobenie výstupnej bitovej informácie. Z toho vyplýva, že maximálny pomer medzi najtmavším a najjasnejším pixelom v snímke bude v prípade 8 bitovej informácie $1 : 256$, resp. $1 : 4096$ pri 12 bitovej informácií. Tento pomer sa nazýva **dynamický rozsah** a definuje sa ako bezrozmerná veličina, ktorá v tomto prípade znamená pomer medzi najsvetlejším a najtmavším pixelom v obraze. Ale čo ak je potrebné zaznamenať scénu s oveľa väčším dynamickým rozsahom?

Odpoveďou je, že pri pokuse o expozíciu takejto scény pomocou digitálneho fotoaparátu vzniknú tzv. preexponované (veľmi svetlé miesta) a podexponované (veľmi tmavé) oblasti. Je to z dôvodu, že hodnoty, ktoré prekročia maximálny dynamický rozsah konkrétneho fotoaparátu, sa reprezentujú ako maximálne (biela farba) alebo minimálne hodnoty (čierna).

Takýto príklad sa nachádza na nasledujúcich dvoch obrázkoch 3.1a a 3.1b. V prípade snímky na ľavej strane vynikli detaily v svetlých miestach, ale takmer všetky detaily v tmavej časti scény zanikli. Naopak v prípade pravej snímky sú vidieť jasne detaily v pôvodne tmavých miestach, ale detaily v oblasti lampičky a pod ňou sú preexponované. Tu sa ukazujú obmedzenia záznamu obrazu s nízkym dynamickým rozsahom (v ďalšom texte je pre takýto obraz používaná skratka „LDR“).

(a) Snímka s krátkou dobou expozície, vynikli svetlé miesta.

(b) Snímka s dlhou dobou expozície, vynikli tmavé miesta.

Obr. 3.1: Na ľavom obrázku je možné vidieť snímku s krátkou dobou expozície a na pravom obrázku dlhšiu dobu expozície (prevzaté z [12]).

Ale čo ak majú byť zachované detaily v rámci celej snímanej scény? Potom je nutné zachytiť celý dynamický rozsah – takéto snímky sa nazývajú HDR.

HDR snímka sa teda definuje ako snímka s veľkým dynamickým rozsahom. Reálny svet okolo nás obsahuje oveľa väčšie dynamické rozsahy ako v predchádzajúcom príklade. V prípade väčšiny dnešných fotoaparátov nie je technicky možné reprezentovať takéto scény na obmedzenom počte bitov pre jednu farebnú zložku. Naproti tomu dokáže ľudské oko vnímať väčší dynamický rozsah svetla, ako sú schopné záznamové zariadenia zachytiť a zobrazovacie zobraziť. Ľudské videnie umožňuje jasne vidieť zrnká piesku na slnkom ožiarenej pláži, ale aj detaily v tmavom kúte pod stolom v silne osvetlenej miestnosti. Ľudské oko sa dokáže prispôsobiť okolitému osvetleniu a svetlo vníma logaritmicky. Aj z tohoto dôvodu vznikla potreba snímania scén v HDR. Hneď sa objavujú otázky:

- Aký zmysel má záznam a spracovanie HDR?
- Ako zachytiť obraz s veľkým dynamickým rozsahom?
- Ako zobraziť obraz s veľkým dynamickým rozsahom?

Využitie HDR obrazu je rozsiahlejšie ako v rámci fotografovania. S HDR je možné sa stretnúť v počítačových hrách (rôzne explózie alebo vesmírne scény s hviezdami na pozadí), nové filmy sú často snímané HDR kamerami, pretože spracovanie takéhoto materiálu umožňuje väčšie možnosti pri tvorbe výsledného filmu. Vstupný obraz v HDR môže takisto poskytovať širšie možnosti pre rôzne typy rozpoznávania alebo klasifikácie. Ako príklad je možné uviesť rôzne sledovacie zariadenia, poprípade sledovanie dopravnej premávky.

3.1.1 Záznam HDR obrazu

Vytvorenie HDR snímky sa v dnešnej dobe realizuje pomocou dvoch základných metód. Prvou možnosťou sú kamery, ktoré sú schopné záznamu takéhoto obrazu. Medzi takéto kamery patrí Panoscan Mark3, iSTAR system alebo SpheronVR. Tieto zariadenia sú síce priamo určené na záznam takéhoto obrazu, ale na druhej strane stojí neporovnateľne vyššia cena v porovnaní s klasickými fotoaparátmi.

Druhou možnosťou je skladanie viacerých LDR snímkov (snímkov s nízkym dynamickým rozsahom). Toto sa dá dosiahnuť pomocou snímania tej istej scény, vždy s inou expozičnou hodnotou. Touto technikou sa dá realizovať záznam HDR aj bežnými fotoaparátmi, čo je nespornou výhodou. Problémom však je nutnosť záznamu scény s čo najmenšími zmenami medzi jednotlivými snímkami, inak vzniká riziko nechcených artefaktov vo výslednom obraze (existujú algoritmy, ktoré tento problém čiastočne eliminujú). Z takejto sekvencie je možné následne zložiť jeden HDR snímok pomocou skladania. Ukážka sekvencie sa nachádza na obrázku 3.2.

Obr. 3.2: Sekvencia LDR snímkov, z ktorých je možné vytvoriť HDR obraz pomocou skladania. Scéna má dynamický rozsah až 469517 : 1 (prevzaté z [12]).

3.1.2 Skladanie HDR obrazu zo sekvencie LDR snímkov

Vytvorenie HDR snímky zo sekvencie LDR snímkov vyžaduje aby bol okrem tejto sekvencie známy aj expozičný čas. Pri zázname je odporúčané mať konštantnú veľkosť clony a hodnotu ISO. Dôvodom je, že veľkosť clony mení aj hĺbku ostrosti a zmena hodnoty ISO sa prejaví zmenou šumu v obraze. Preto je odporúčané meniť iba expozičný čas. Minimálny počet snímok je aspoň 3, kde každá má inú dĺžku expozície. Ako je možné vidieť na sekvencií, ktorá je na obrázku 3.2, na snímkach s krátkym časom expozície vynikli detaily v svetlých častiach obrazu a na snímkach s dlhším časom expozície vynikli pôvodne tmavé miesta. Tieto snímky sa preložia cez seba a z odpovedajúcich pixelov sa spočítajú hodnoty pre HDR snímok, preto sa často táto technika označuje Per-Pixel mapovanie.

Pre vytvorenie kvalitného HDR obrazu je dôležité zaistiť, aby bola zachytená vždy tá istá scéna. Problémom však je samotný pohyb fotoaparátu, ktorý je spôsobený aj minimálnym pohybom fotoaparátu (riešením môže byť použitie statívu a automatického snímania

viacerých snímok s rôznou hodnotou EV). Takisto je podstatné, aby samotná scéna bola statická, ak to nie je dodržané, tak sa vo výslednom obraze objavia tzv. duchovia.

Vytvorenie samotnej HDR snímky vyžaduje ešte váhovú funkciu, ktorá priraďuje váhy jednotlivým pixelom LDR. Pomocou tejto funkcie je možné priradiť intenzitám pixelov rôznych snímiek váhy, akými ovplyvňujú výslednú hodnotu pixelu. Týmto spôsobom je dosiahnuté toho, aby vo výsledku preexponované a podexponované pixely prispievali k výslednej hodnote iba malou váhou. Pre výpočet jedného pixelu HDR snímky teda platí (prebrané z [13]):

$$L_{i,j} = \frac{\sum_{k=1}^{N} \frac{f^{-1}(Z_{k_{i,j}}) \cdot w(Z_{k_{i,j}})}{\Delta t_k}}{\sum_{k=1}^{N} w(Z_{k_{i,j}})} \tag{3.1}$$

Kde $L_{i,j}$ je pixel HDR snímky, f^{-1} vyjadruje inverznú funkciu odozvy snímača, $Z_{k_{i,j}}$ je pixel LDR snímky, $w(Z_{k_{i,j}})$ vyjadruje váhovú funkciu pre pixel, Δt_k je doba expozície a N je počet LDR snímkov.

3.1.3 Formáty pre reprezentáciu HDR

Pre reprezentovanie takýchto snímkov však nie je vhodná reprezentácia ako pri klasických snímkach, kde najčastejšie používané sú snímky s 8 bitovou farebnou hĺbkou. To znamená, že jeden farebný kanál je schopný reprezentovať 2^8 hodnôt, čo je však nedostatočné pre reprezentovanie HDR obrazu. Existuje viacero formátov, ktoré sú určené pre uloženie. Spoločným znakom je, že jednotlivé bity nie sú reprezentované celými číslami, ale pomocou čísel s pohyblivou rádovou čiarkou alebo väčšou bitovou hĺbkou. Medzi takéto formáty patrí:

- OpenEXR formát (prípona `.exr`), ktorý podporuje 32 a 16 bitový bitový formát s pohyblivou rádovou čiarkou a 32 bitový formát s celými číslami.
- Radiance (`.pic`, `.hdr`) formát využíva iný spôsob reprezentácie. Jeden RGB pixel je reprezentovaný ako 32 bitov, kde každý kanál RGB sa reprezentuje na 8 bitoch a zvyšných 8 bitov predstavuje spoločný exponent.
- LogLuv TIFF je formát, ktorý umožňuje uchovávať HDR snímky v TIFF súbore. Tento formát ale uchováva informáciu o snímke odlišným spôsobom. Jeden pixel je reprezentovaný na 32 bitoch. LogLuv využíva 16 bitov na zakódovanie jasu v pevnej rádovej čiarke. Tento jas sa upraví logaritmom so základom 2. Zvyšných 16 bitov je určených pre reprezentáciu farby.
- Portable float map (`.pfm`) reprezentuje každý RGB pixel ako trojicu hodnôt v plávajúcej rádovej čiarke.

3.1.4 Zobrazenie HDR

V predchádzajúcej podkapitole bolo ukázané, ako vytvoriť a reprezentovať HDR snímok v pamäti. Avšak teraz nastáva problém zobrazenia takéhoto obrazu. Dnešné zobrazovacie zariadenia, napríklad displeje alebo tlačiarne, nemajú dostatočný dynamický rozsah na zobrazenie takéhoto obrazu. Tak ako v prípade vytvárania snímok, aj tu je viacero možností pre zobrazenie.

Prvou možnosťou sú špeciálne LCD displeje, ktoré sú určené práve pre HDR. Problémom je stále ich vysoká cena, pretože zvyšovanie dynamického rozsahu v bežne dostupných displejoch (či už televízie alebo monitory) v posledných rokoch stálo v úzadí za napredovaním vo zvyšovaní rozlíšenia. Avšak s nástupom rozlíšenia označovaného ako 4K (4096×2160) sa objavuje čoraz viac produktov, ktoré umožňujú zobraziť vysoký dynamický rozsah. Spoločnosti Sony alebo Samsung predávajú televízory a displeje, ktoré umožňujú zobrazovať väčší dynamický rozsah ako dnes bežne dostupné produkty. Je teda možné očakávať veľký rozmach HDR v najbližších rokoch.

Druhá možnosť je založená na kompresií jasu v obraze. Pre zobrazenie na klasickom displeji teda musí byť dynamický rozsah znížený na taký, aký má zobrazovacie zariadenie. Naivné prístupy ako lineárne namapovanie do rozsahu $0 - 255$ sú nepostačujúce, pretože cieľom je zachovať vo výslednom obraze čo najviac informácií. Z tohoto dôvodu bolo vykonaných viacero výskumov, ktoré sa venovali tejto problematike. Konkrétne sa tento proces nazýva mapovanie tónov alebo mapovanie tonality.

3.2 Mapovanie tónov

Mapovanie tónov je teda operácia, v ktorej sa jas v HDR obraze premapuje na obraz s nižším dynamickým rozsahom, pričom takýto obraz už je možné zobraziť na zobrazovacom zariadení. Tento postup sa často nazýva ako mapovací operátor – Tone Mapping Operator (TMO). V posledných dvadsiatich rokoch vzniklo veľké množstvo takýchto operátorov, pretože zvládnutie tejto problematiky je obtiažne dosiahnuteľné pre všetky typy scén. Využitých je viacero prístupov, ktoré využívajú rôzne poznatky od anatómie, počítačového videnia, spracovania obrazu až po psychologické výskumy. Niektoré sa orientujú na realistické zobrazenie pôvodnej scény, inými je možné zvýrazniť určité informácie v obraze, či napodobňovanie princípu ľudského videnia. Avšak stále je ich možné klasifikovať podľa rôznych kritérií. Najcharakteristickejšou je klasifikácia na lokálne a globálne operátory, ktoré rozdeľujú metódy podľa vlastnosti výpočtu hodnoty pixelu:

- **Globálne operátory** (je možné sa stretnúť s názvom priestorovo jednotné, z anglického „spatially uniform“) sú charakteristické tým, že pre všetky pixely v obraze využívajú rovnakú nelineárnu mapovaciu funkciu a využívajú iba samotnú hodnotu pixelu, tj. bez využitia hodnôt okolitých pixelov. Všeobecne sú výpočtovo menej náročné ako lokálne operátory. Tieto operátory sú pomenované podľa svojich autorov a patria sem napríklad operátory od Tumblina a Rushmeiera, Reinhardov operátor, Wardov operátor, Pattanaik a spol.

- **Lokálne operátory** (alebo priestorovo nejednotné, z anglického „spatially varying“) pre výpočet pixelov využívajú aj okolie pixelu. Pre všetky operátory je možné povedať, že pre každý pixel má výpočet nového jasu tvar: $L(x, y) = S(x, y) \cdot L_w(x, y)$, kde $L_w(x, y)$ je jas počítaného pixelu a je vynásobený hodnotou funkcie $S(x, y)$, ktorá je vypočítaná z okolia pixelu. Patria sem napríklad Chiu a spol., Reinhardov operátor alebo Durand a Dorsey.

V nasledujúcich podkapitolách je pozornosť venovaná detailnému popisu zvolených operátorov pre implementáciu. Z globálnych sú zvolené metódy Dragov operátor a Reinhardov operátor. Z lokálnych je zvolenou metódou operátor od pánov Duranda a Dorseyho.

3.3 Dragov globálny operátor

Prvým implementovaným operátorom bude globálny operátor, ktorý bol prezentovaný v práci pána F. Draga, K. Myszkowského, T. Annenena a N. Chiba v roku 2003. Najčastejšie je možné sa stretnúť s označením `Drago03`. Bol zvolený z dôvodu jednoduchosti, ktorá umožní rýchlu implementáciu a otestovanie vlastností Vivado HLS. Nasledujúci popis vychádza z originálnej publikácie [14].

3.3.1 Algoritmus

Predpokladaným vstupom je HDR obraz vo formáte RGB. V prvom kroku je prevedený formát RGB na formát CIE 1931 Yxy. Prevod je definovaný nasledovne: Najskôr je farebný formát RGB transformovaný do farebného priestoru CIE 1931 XYZ pomocou rovnice 3.2 a následne do Yxy prostredníctvom rovníc 3.3 a 3.4 (koeficienty matice odpovedajú hodnotám pre prevod z farebného formátu sRGB).

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0173 & 0.1192 & 0.9505 \end{bmatrix} . \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.2}$$

$$W = X + Y + Z \tag{3.3}$$

$$Y = Y, \qquad x = \frac{X}{W}, \qquad y = \frac{Y}{W} \tag{3.4}$$

Kde R, G, B sú intenzity vo farebnom priestore RGB a X, Y, Z sú intenzity vo farebnom priestore XYZ. W vyjadruje váhu pre následny prevod do priestoru Yxy. Výsledné hodnoty Y, x, y v rovnici 3.4 sú finálne intenzity vo farebnom priestore Yxy.

Najdôležitejší pre ďalší výpočet je kanál Y, ktorý reprezentuje jas pixelu (X,Z kanály sú nositeľmi farby). Následne sa spočítajú globálne informácie o obraze, ktoré sú potrebné pre tónové mapovanie:

- Maximálna hodnota z kanála Y – Y_{max}
- Logaritmický priemer scény na základe hodnôt jasu všetkých pixelov. Výpočet vychádza z operátoru, ktorý publikovali páni Tumblin a Rushmeier. Ide o globálny príznak označovaný ako „World adaption luminance“ a bude značený ako Y_{wavg}. Výpočet sa nachádza v rovnici 3.5.

$$Y_{wavg} = \frac{\sum_{i=1}^{h} \sum_{j=1}^{w} \log(Y_{i,j} + 2.3 \times 10^{-5})}{w.h} \tag{3.5}$$

  Kde w, h sú rozmery vstupnej snímky, Y je kanál jasu, a hodnota 2.3×10^{-5} je konštanta, ktorú definoval pán Tumblin vo svojej práci.

Teraz už je možné realizovať samotný tónovací operátor, ktorý je v základe jednoduchý. Vstupom do mapovacej funkcie sú nasledovné parametre:

- kanál Y
- Spočítané globálne parametre – maximálna (Y_{max}) a priemerná (Y_{wavg}) hodnota jasu

- Bias parameter (značené b) – užívateľom definovaný parameter, ktorý určuje ako je mapovaný jas, má veľký vplyv na výsledok. Odporúčaná hodnota je $< 0.70, 0.90 >$.
- Hodnota expozície – užívateľom definovaný parameter, základná hodnota je 1.

V rovnici 3.6 je ukázaný výpočet novej hodnoty Y.

$$Y_{new_{i,j}} = \frac{L_{dmax} \cdot 0.01}{\log_{10}(Y_{wmax} + 1)} \cdot \frac{\log(Y_{w_{i,j}} + 1)}{\log(2 + ((\frac{Y_{w_{i,j}}}{Y_{wmax}})^{\frac{\log(b)}{log(0.5)}}))} \tag{3.6}$$

Kde $Y_{new_{i,j}}$ je nová intenzita Y kanálu na pozícií i, j. L_{dmax} vyjadruje faktor pre prispôsobenie výstupného displeja (štandardne $100cd \cdot m^{-2}$, čiže hodnota 100). Y_{wmax} je maximálna hodnota Y kanálu váhovaná Y_{wavg}. Premenná $Y_{w_{i,j}}$ vyjadruje vstupnú hodnotu, ktorá je váhovaná Y_{wavg} a vynásobená hodnotou expozície, spolu teda: $Y_{w_{i,j}} = \frac{Y_{i,j}}{Y_{wavg}} \cdot expo$.

Pri implementácií je možné ako základnú optimalizáciu využiť faktu, že časti výrazu pre výpočet novej hodnoty jasu je možné vypočítať pred samotným cyklom výpočtu nových intenzít. Zjednodušený zápis výpočtu sa nachádza na nasledujúcom pseudo algoritme 3.1.

```
1  function drago03( channelY, maxY, worldLum, BIAS, exposition )
2  begin
3    avLum=exp(worldLum)
4    biasP = log(biasParam)/LOG05
5    contP = 1/contParam
6    Lmax = maxY/avLum
7    divider = log10(Lmax+1)
8
9    for all values Y of channelY
10   begin
11     Y=Y/avLum
12     Y=Y*exposition
13     interpol = log (2 + bias(biasParam, Y / maxY) * 8)
14     Y=((log(Y+1)/interpol)/divider)
15   end
16 end
17
18 function bias( b,  x)
19 begin
20   return pow (x, b);
21 end
```

Algoritmus 3.1: Pseudoalgoritmus dragovho mapovacieho operátora.

Posledným krokom je prevod z Yxy späť do RGB. Proces prebieha inverzným spôsobom a najskôr sa z Yxy spočítajú hodnoty farebného modelu XYZ podľa vzorca, ktorý je v rovnici 3.7. V poslednom kroku treba vykonať prevod podľa rovnice 3.8 (koeficienty matice odpovedajú hodnotám pre prevod do farebného formátu sRGB). Týmto krokom je ukončený výpočet.

$$X = \frac{(x.Y)}{z}, \qquad Y = Y, \qquad Z = \frac{X}{x} - X - Y; \tag{3.7}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.2405 & -1.5371 & -0.4985 \\ -0.9693 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0572 \end{bmatrix} . \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.8}$$

3.4 Reinhardov operátor

Druhý implementovaný operátor je publikovaný v práci, ktorej autori sú Erik Reinhard a Kate Devlin. Najčastejšie je označovaný ako `Reinhard02`. Ide o zložitejší globálny operátor, ktorý využíva viac informácií o vstupnom obraze, čo by malo v konečnom dôsledku znamenať kvalitnejší výsledok. Operátor je inšpirovaný ľudským videním a operátor je možné parametrizovať pomocou troch parameterov. Tieto parametre majú simulovať adaptáciu fotoreceptoru. Nasledujúci popis vychádza z pôvodnej publikácie [15].

3.4.1 Algoritmus

Tak ako pri operátore `Drago03` aj v tomto prípade je vstupom obraz vo formáte RGB. Narozdiel od neho sa v prvom kroku extrahuje iba jasová zložka vstupného obrazu pomocou rovnice 3.9. Výraz je rovnaký ako pri výpočte hodnoty kanála Y vo farebnom priestore CIE XYZ, ale kanál je značený písmenom L.

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B \tag{3.9}$$

Kde L je intenzita jasu a R, G, B sú intenzity odpovedajúcich kanálov vo farebnom priestore RGB.

Ďalej sa spočítajú parametre vstupného obrazu:

- Maximálnu hodnotu jasu L_{max}
- Minimálnu hodnotu jasu L_{min}
- Priemer jasu všetkých pixelov L_{av}
- Logaritmický priemer jasu scény na základe hodnôt jasu všetkých pixelov, ktorý sa spočíta rovnako ako pri `Drago03` (rovnica 3.5) – L_{world}
- Priemer hodnôt pixelov každého kanálu RGB R_{av}, G_{av}, B_{av}

Tieto hodnoty sú následne prepočítané na parametre pre mapovanie:

$$L_{max} = log(L_{max}) \tag{3.10}$$

$$L_{min} = log(L_{min}) \tag{3.11}$$

Kľúč snímky, ktorý globálne popisuje danú scénu na základe hodnôt jasu:

$$k = \frac{(L_{max} - L_{world})}{(L_{max} - L_{min})} \tag{3.12}$$

Kontrast snímky, ktorý je závislý na hodnote kľúča snímky:

$$m = 0.3 + 0.7.k^{1.4} \tag{3.13}$$

Jas snímky:

$$f = exp(-br) \tag{3.14}$$

Tónovací operátor je parametrizovaný pomocou hodnot f, a, c:

- f umožňuje riadiť celkový jas obrazu
- a určuje vplyv lokálnej adaptácie na mapovanie
- c riadi korekciu farieb

V kroku mapovania tónov je najskôr potrebné spočítať hodnoty lokálnej I_l a globálnej adaptácie I_g. Výpočet lokálnej adaptácie I_l sa nachádza v rovnici 3.15 a globálnej v 3.16. Pre zjednodušenie zápisu sú nasledujúce rovnice pre jeden farebný kanál. V reálnej implementácií je potrebné realizovať výpočet pre každý farebný kanál samostatne.

$$I_{l_{i,j}} = c \cdot H_{i,j} + (1 - c) \cdot L_{i,j} \tag{3.15}$$

Kde $I_{l_{i,j}}$ je lokálna adaptácia pixelu na pozícií (i, j), c je parameter korekcie farieb, $H_{i,j}$ je vstupný pixel HDR snímky na pozícií (i, j), $L_{i,j}$ je odpovedajúci jas pixelu.

$$I_{g_{i,j}} = C_{av} + (1 - c) \cdot L_{av} \tag{3.16}$$

Kde $I_{g_{i,j}}$ je globálna adaptácia pixelu na pozícií (i, j), c je parameter korekcie farieb, C_{av} je priemerná hodnota intenzít odpovedajúceho kanálu, L_{av} je priemerná hodnota jasu.

Teraz je možné spočítať interpolovanú svetelnú adaptáciu pomocou rovnice 3.17:

$$I_{a_{i,j}} = a \cdot I_{l_{i,j}} + (1 - a) \cdot I_{g_{i,j}} \tag{3.17}$$

Kde $I_{a_{i,j}}$ interpolovaná svetelná adaptáciu, a je vstupný parameter, $I_{l_{i,j}}$ je odpovedajúca lokálna adaptácia, $I_{g_{i,j}}$ je odpovedajúca globálna adaptácia.

Po týchto krokoch je možné vykonať mapovanie tónov:

$$O_{i,j} = \frac{H_{i,j}}{H_{i,j} + (f \cdot I_a)^m} \tag{3.18}$$

Kde $O_{i,j}$ je nová intenzita pixelu na pozícií (i, j), $H_{i,j}$ je odpovedajúca hodnota v HDR snímke, f je vstupný parameter, $I_{a_{i,j}}$ je odpovedajúca interpolovaná adaptácia.

V poslednom kroku sú spočítané maximálne a minimálne hodnoty farieb, pomocou ktorých je vykonaná normalizácia do intervalu $< 0, 1 >$.

3.5 Lokálny operátor Durand a Dorsey

V roku 2002 autori Frédo Durand a Julie Dorsey publikovali v publikácií [16] jeden z najznámejších tónovacích operátorov. Vychádza z predpokladu, že jasová informácia, ktorá je zodpovedná za vysoký kontrast, sa mení pomaly a detailná jasová informácia sa mení rýchlo. Durand a Dorsey sa rozhodli rozdeliť obraz do troch vrstiev – základná, detailná a vrstva intenzít („radiance map“). Prvou je tzv. **základná vrstva**, ktorá obsahuje jasovú informáciu a je vytvorená pomocou špeciálneho filtru na vstupné intenzity v obraze. Pre tento účel sa používa vyhladzovací filter, ktorý sa nazýva bilaterálny. Druhou je tzv. **detailná vrstva**, ktorá vznikne ako diferencia medzi základnou vrstvou a **vrstvou intenzít**. Následne je komprimovaný jas v obraze. Nasledujúci popis vychádza okrem originálnej publikácie aj z implementácie v knižnici **pfstools**, ktorá zahŕňa súčasný stav poznania v oblasti HDR [1]. Dôvodom je, že originálna implementácia obsahuje operácie, ktoré sú problematicky syntetizovateľné.

3.5.1 Algoritmus

V prvom kroku, podobne ako pri prechádzajúcich operátoroch, je potrebné extrahovať kanál, ktorý nesie informáciu o jase. V tomto prípade je použitý nasledujúci vzorec pre výpočet intenzít:

$$I_{i,j} = \frac{1}{61} \cdot (20 \cdot R_{i,j} + 40 \cdot G_{i,j} + B_{i,j}) \tag{3.19}$$

Kde I je intenzita pixelu a R, G, B sú intenzity odpovedajúcich kanálov vo farebnom priestore RGB. V ďalšom kroku je nutný priechod celým vstupným obrazom, pri ktorom sú kanály RGB váhované odpovedajúcou hodnotou intenzity:

$$R_{i,j} = \frac{R_{i,j}}{I_{i,j}}, \qquad G_{i,j} = \frac{G_{i,j}}{I_{i,j}}, \qquad B_{i,j} = \frac{B_{i,j}}{I_{i,j}}, \qquad I_{i,j} = \ln\left(I_{i,j}\right) \tag{3.20}$$

Nasledujúci krok sa týka vrstvy intenzít, na ktorú je aplikované bilaterálne filtrovanie. V originálnom dokumente je popisovaný rýchly bilaterálny filter, ale je možné využiť aj klasickú implementáciu, ktorá je jednoduchšia. Tento filter je možné definovať ako filter, ktorý odstraňuje šum v obraze, ale zachováva ostré hrany. Využíva 2D konvolučnú masku, ktorá je závislá na hodnote okolitých pixelov. Pre výpočet jedného pixelu platí (nasledujúci popis vychádza z [17]):

$$BF_p[I] = \frac{1}{W_q} \sum_{q \in S} G_{\sigma_r}(||p - q||) \cdot G_{\sigma_s}(|I_p - I_q|) \cdot I_q \tag{3.21}$$

A normalizačný faktor W_q je spočítaný ako:

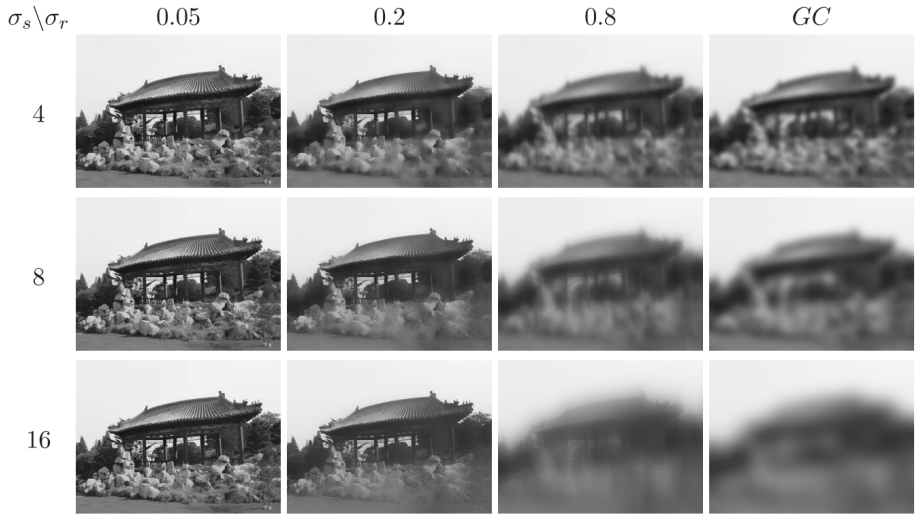
$$W_q = \sum_{q \in S} G_{\sigma_r}(||p - q||) \cdot G_{\sigma_s}(|I_p - I_q|) \tag{3.22}$$

Kde $G_{\sigma_s}(||p-q||)$ je zápis Gaussového filtru, ktorý určuje novú hodnotu intenzity pixelu ako vážený priemer susedných intenzít, pričom váha klesá s rastúcou vzdialenosťou od pixelu p, ktorý je v strede masky. Váha pixelu q závisí iba od vzdialenosti medzi pixelmi a nie od

[1] Knižnica dostupná na `http://pfstools.sourceforge.net/pfstmo.html`

ich vlastných intenzít, z toho vyplýva, že jasný pixel má vyššiu váhu ako tmavý, aj keď sú rovnako vzdialené od pixelu v strede masky. Druhá časť $G_{\sigma_r}(|I_p - I_q|)$ vyjadruje hodnotu „podobnosti“ intenzity pixelu v strede konvolučnej masky a pixelu v okolí. To umožní aby pixely s podobnou hodnotou intenzity mali väčšiu váhu než rozdielne.

Filter je možné parametrizovať pomocou hodnôt σ_s a σ_r. S väčšou hodnotou σ_r sa stráca efekt zachovania hrán, pretože vzdialené pixely majú väčšiu váhu a tým viac filtrovanie pripomína Gaussovské rozostrenie. Zvyšovaním hodnoty σ_s filter rozmazáva čoraz väčšie objekty. Efekt týchto parametrov je ukázaný na obrázku 3.3.



Obr. 3.3: Obrázok ukazuje závislosť bilaterálneho filtrovania na hodnotách σ_s a σ_r. S rastúcou hodnotou σ_r je vidieť väčší efekt rozmazania obrazu. S rastúcou hodnotou σ_s je vidieť rozmazávanie čoraz väčších objektov (prevzaté z [17]).

Aplikovaním tohoto filtru na vypočítané intenzity dostávame novú vrstvu, ktorá sa nazýva **základná vrstva** (ďalej značená ako B). Následne je potrebné vyhľadať minimálnu a maximálnu hodnotu tejto vrstvy. Teraz je možné vyjadriť tzv. kompresný faktor, ktorý bude ďalej značený ako `CF` a výpočet je zobrazený v rovnici 3.23. Hodnota `baseContrast` je užívateľom definovaná, optimálne výsledky sú dosiahnuté ak sa rovná 4 až 5.

$$CF = \frac{baseContrast}{B_{max} - B_{min}} \tag{3.23}$$

Predposledným krokom je výpočet tzv. detailnej vrstvy, ktorá vznikne ako rozdiel vrstvy intenzít základnej vrstvy, tento výpočet je v rovnici 3.24. Následne sa spočítajú nové hodnotu intenzity pomocou rovnice 3.25.

$$D_{i,j} = I_{i,j} - B_{i,j} \tag{3.24}$$

$$I_{i,j} = B_{i,j}.CF + D_{i,j} \tag{3.25}$$

Nakoniec sa pre kanály RGB všetkých pixelov prepočíta ich intenzita:

$$R_{i,j} = R_{i,j}.\exp(I_{i,j}), \quad G_{i,j} = G_{i,j}.\exp(I_{i,j}), \quad B_{i,j} = B_{i,j}.\exp(I_{i,j}) \tag{3.26}$$

Kapitola 4

Špecifikácia zadania

V tejto kapitole sú stanovené ciele diplomovej práce a popísané použité prostriedky pre implementáciu. Aplikácia bude bežať na platforme Xilinx Zynq a spracovávať obraz z kamery, ktorý prostredníctvom zbernice predá FPGA komponentom na spracovanie. Toto spracovanie bude prebiehať v reálnom čase a zahŕňa aplikovanie operátoru mapovania tonality na HDR obraz. Po spracovaní sa obraz odovzdá HDMI komponentu, ktorý výsledok zobrazí na monitore.

4.1 Stanovenie cieľov

Cieľom diplomovej práce je implementácia a otestovanie aplikácie, ktorá bude realizovať mapovanie tónov v HDR snímkach. Bude implementovaných viacero metód mapovania tónov: Dragov operátor, Reinhardov operátor a lokálny operátor Durand a Dorsey. Aplikácia musí vhodným spôsobom využiť procesorový systém aj programovateľnú logiku. Procesorový systém bude realizovať príjem obrazu z kamery, ktorá je na obrázku 4.1 a prostredníctvom zbernice AXI ho predá do programovateľnej logiky. Programovateľná logika je vyhradená pre realizovanie mapovania tónov, po ktorej nasleduje predanie dát späť do pamäte s možnosťou zobrazenia výstupu na HDMI monitore.

Obr. 4.1: Zapožičaná kamera, ktorá je využitá na snímanie viacerých expozícií.

4.1.1 HW a SW prostriedky

Pre realizáciu finálnej aplikácie som obdržal vývojový kit Xilinx Zynq ZC702 a multi–expozičnú, farebnú HD kameru od spoločnosti Camea. Pre implementáciu komponentov mapovania tónov je zvolený nástroj Vivado HLS vo verzií 2015.1. Návrh celkového dizajnu aplikácie bude vytvorený v prostredí Vivado vo verzií 2015.1. Zo softwarových knižníc bude využitá softwarová knižnica OpenCV, ktorá umožňuje prácu s obrazom a videom. Konkrétne ide o verziu 2.4.7, ktorá je nainštalovaná na vývojovom kite. Ďalej som obdržal IP jadro, ktoré vykonáva transformáciu Axi Video Streamu na HDMI rozhranie a ovládač pre kameru.

Kapitola 5

Implementácia

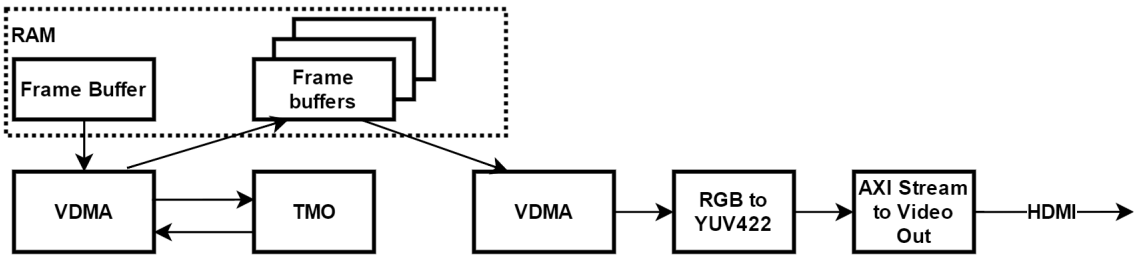
Kapitola je venovaná návrhu a samotnej implementácií operátorov za využitia Vivado HLS. Na začiatku kapitoly je navrhnutá základná architektúra aplikácie a následne sú popísané implementované operátory. Kapitola pokračuje optimalizáciou operátorov, vyhodnotením implementácie a popisom finálnej aplikácie pre Xilinx Zynq. Pre testovanie operátorov sú využité snímky vo formáte OpenEXR, ktoré je schopné načítať knižnica OpenCV. Tieto snímky sa nachádzajú na priloženom CD.

5.1 Návrh aplikácie

Použitá platforma Xilinx Zynq, ako bolo popísané v kapitole 2.5, poskytuje procesorový systém a programovateľnú logiku. Je teda vhodné aby bola aplikácia rozdelená medzi tieto dve časti. Takisto je vhodné využiť AXI VDMA (Video DMA), čo umožní predať zodpovednosť prenosu dát jednotke VDMA a protokolu AXI.

5.1.1 Architektúra aplikácie

Základná architektúra je zobrazená na obrázku 5.1, kde je možné vidieť postupnosť spracovania obrazu. Procesorový subsystém sa stará o načítavanie a predspracovanie videa, ktoré následne uloží do vyhradeného miesta v pamäti (na obrázku Frame Buffer), kde riadenie preberie AXI VDMA a prostredníctvom zbernice transferuje video do programovateľnej logiky, kde sa nachádza akcelerovaný FPGA komponent pre mapovanie tónov. Po spracovaní VDMA cyklicky ukladá snímky do vyrovnávacích pamätí pre HDMI výstup (na obrázku Frame Buffers). Druhý VDMA komponent následne odovzdáva video stream do komponentu realizujúceho prevod do farebného formátu YUV422. Výhodou je, že takto môžu pracovať subsystémy nezávisle na sebe a s rôznou pracovnou frekvenciou.



Obr. 5.1: Návrh základnej architektúry implementovanej aplikácie.

Komponent mapovania tónov bude implementovaný v prostredí Vivado HLS a zvoleným jazykom je C++. Pre konfigurovanie komponent v reálnom čase, bude využitý AXI4-Lite protokol, ktorý je vhodný na nízko rýchlostné dátové toky.

5.1.2 Špecifikácia vstupných a výstupných dát

Aby bolo možné vytvoriť jednotné rozhranie komponent, budú mať všetky 3 implementované metódy na vstupe a výstupe AXI Video Stream so šírkou 96 bitov. Obsahom sú hodnoty pixelov vo formáte RGB, kde dátovým typom je `float`. Toto uloženie je zhodné s uložením dát v pamäti, ktoré využíva knižnica OpenCV, čo umožní jednoduchšie prepojenie medzi SW a FPGA komponentami. Štruktúra je nasledovná a je zhodná pre vstupný aj výstupný stream:

- bity $< 0, 31 >$ – intenzita kanálu B
- bity $< 32, 63 >$ – intenzita kanálu G
- bity $< 64, 96 >$ – intenzita kanálu R

5.1.3 Rozhranie implementovaných komponent

Všetky implementované metódy majú ekvivalentné rozhranie, ktoré obsahuje vstupný a výstupný AXI Video Stream a AXI-Lite rozhranie, ktorým je sprístupnená konfigurácia parametrov jednotlivých metód (jediný rozdiel je v type a počte parametrov, ktoré sú mapované v pamäti). Top-level funkcie sú pomenované podľa ich tvorcov a to v tvare `TMO_meno00`, konkrétne `TMO_Drago03`, `TMO_Reinhard05` a `TMO_Durand02`. Takto je dosiahnuté vlastnosti, že stačí zameniť tónovaciu komponentu v dizajne a vysyntetizovať bez ďalších zmien.

5.2 Dragov operátor

V prvom kroku bola vytvorená implementácia v jazyku C++ s využitím knižnice OpenCV. Pri implementácií sa vychádzalo z oficiálnej práce autorov. Tento krok umožnil lepšie pochopenie vlastností operátora a jednoduchšiu implementáciu v prostredí Vivado HLS. SW implementácia sa nachádza na priloženom CD v priečinku `Sources/SW/drago02`.

5.2.1 Implementácia vo Vivado HLS

Hlavným cieľom je vytvoriť komponentu, ktorá má pracovať na frekvencií 100 MHz a bude sa dať zreťaziť s inicializačným intervalom $II = 1$. Vzhľadom k popísanému algoritmu je potrebné sa vysporiadať s problémom dvoch priechodov cez obraz. Pri prvom priechode snímky je extrahovaná jasová zložka a spočítaná logaritmická suma jasovej zložky. V druhom priechode je vykonané samotné mapovanie na novú hodnotu jasu. To je však v prípade implementácie v FPGA problém, keďže uloženie snímky v BlockRAM na FPGA nie je realizovateľné, pretože nemá dostatočnú pamäť pre uloženie celej snímky. Pre HD snímku je potrebné: $1280 \times 720 \times 3 \times sizeof(float) = 10,5$ MB, ZC 702 má 280×18 Kb BRAM $= 0,63$ MB.

Navrhované sú dve riešenia, kde každé poskytuje isté výhody a nevýhody:

- Rozdelenie výpočtu na dve nezávislé časti – prvá bude obraz transformovať z farebného priestoru RGB do farebného priestoru Yxy a počítať globálne príznaky. Výsledok

uloží späť do pamäte RAM, kde ho následne využije druhá časť, ktorá bude realizovať samotné mapovanie tónov s využitím výstupu prvej časti. Nevýhodou je, že sú potrebné dve DMA, kde každá obsluhuje jeden komponent. Výhodou je možnosť vytvoriť zreťazenie na úrovni komponent.

- Druhou možnosťou je mať iba jeden komponent, ktorý bude realizovať oba priechody, ale trochu rozdielnym spôsobom. Pre aktuálnu vstupnú snímku bude počítať aktuálne globálne príznaky jasu, ale pre mapovanie budú využité príznaky z predchádzajúcej spracovanej snímky. Je zrejmé, že v prípade dvoch nesúvisiacich snímok by dochádzalo k chybnému výpočtu, ale v prípade videa, kde sú snímky podobné, bude chyba malá. Pre prípad spracovania jednej snímky stačí vykonať dva priechody tej istej snímky. Výhodou je šetrenie zdrojov, keďže stačí jedno DMA a jeden komponent na realizovanie kompletnej funkčnosti mapovania tónov.

Pre implementáciu bola zvolená druhá možnosť, ktorá pre výpočet aktuálnej snímky využíva parametre predchádzajúcej. Implementácia vychádza z algoritmu popísaného v podkapitole 3.3. V implementácií je využitý ako hlavný dátový typ `float`, ktorý síce zaberie viac zdrojov, ale umožní na začiatok rýchlejšie ladenie a porovnanie výsledkov s implementáciou pre procesor.

Pre čítanie zo streamu je využitá blokujúca funkcia `read( )`. Takto sú po prečítaní k dispozícií dáta, ktoré je treba uložiť do premennej typu `float`. Ukázalo sa, že priame priradenie rozsahu bitov spôsobuje konverziu na celočíselný dátový typ. Je potrebné použiť variantnú dátovú štruktúru, ktorá umožní prevod z rozsahu bitov na premennú typu `float`. Táto štruktúra obsahuje dve položky – dátový typ `unsigned int` a `float`. Najskôr sú prijaté hodnoty zo streamu uložené ako `unsigned int` a potom prečítané ako `float`.

Samotné mapovanie tónov je realizované identicky ako je popísané v kapitole 3.3, ktorá obsahuje teoretický rozbor operátoru. Malou úpravou je nahradenie delenia za násobenie prevrátenou hodnotou v prípade premenných, ktoré sú konštantné v rámci výpočtu jednej snímky (normovaná maximálna hodnota jasu – `lumMaxNorm` a finálny deliteľ – `divider`). Problémom je však reprezentovanie mocniny, ktorá nie je priamo syntetizovateľná pomocou knižnice `hls_math.h`. To je vyriešené nasledovným vzťahom:

$$x^y = e^{y \cdot \ln(x)} \tag{5.1}$$

5.2.2 Výsledky syntézy

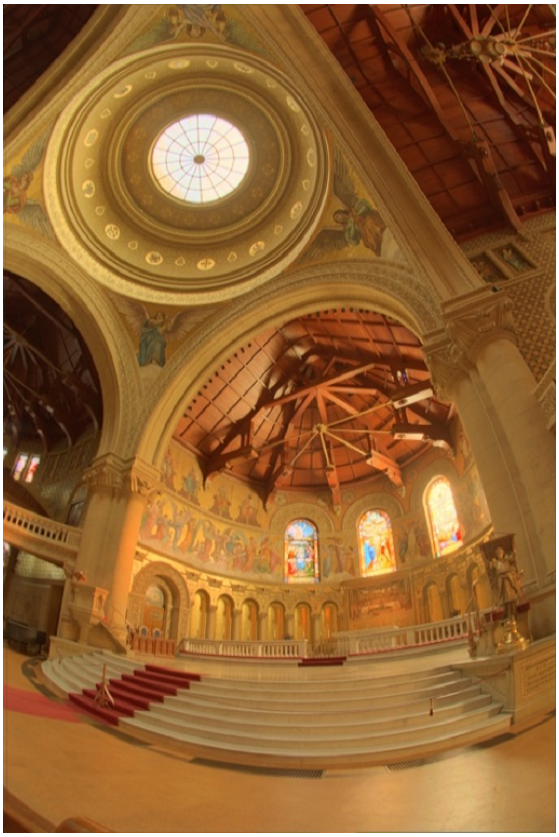
V tabuľke 5.1 sú zobrazené výsledky syntézy pre rôzne pracovné frekvencie, dizajn nie je zreťazený. Najviac zdrojov je spotrebovaných na vytvorenie násobičiek a deličiek, čo je spôsobené samotným princípom násobenia a delenia v plávajúcej rádovej čiarke.

Rýchlosť [MHz]	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
20	0	66	4169	11426
50	0	57	4444	10153
100	0	57	6092	10789
200	0	57	9942	12479

Tabuľka 5.1: V tabuľke je zobrazený výsledok syntézy s rôznymi pracovnými frekvenciami a bez zreťazenia (ide o odhad využitia po procese HLS).

5.2.3 Ukážky

Nasledujú ukážky výstupov implementácie v FPGA (obrázky 5.2a a 5.2b). Tieto snímky boli zvolené z galérie, ktorá je k dispozícií ku knižnici `open-exr` (Snímky sa nachádzajú na priloženom CD v zložke Images). Je vidieť, že pôvodne zaznamenaný obraz je reprodukovaný správne – jas v oknách je vyšší ako v ostatných oblastiach. V prípade snímky `memorial.exr` sú zachované aj detaily na mozaikách okien.



(a) `memorial.exr` po aplikovaní Dragovho operátora.

(b) `designCenter.exr` po aplikovaní Dragovho operátora.

Obr. 5.2: Ukážka tónového mapovania pomocou dragovho operátoru. Bias parameter je nastavený na hodnotu 0.85 a hodnota expozície je 1.

5.3 Reinhardov operátor

Tak ako v prípade Dragovho operátoru, aj v tomto bola najskôr vytvorená referenčná implementácia v jazyku C++ s využitím OpenCV. Vychádza z algoritmu, ktorý je popísaný v pôvodnom dokumente. Táto implementácia je tiež na priloženom CD v zložke `Sources/SW/reinhard05`.

5.3.1 Implementácia vo Vivado HLS

Podobne ako pri Dragovom operátore aj pri Reinhardovom je cieľom vytvoriť komponent, ktorý bude schopný pracovať na frekvencií 100 Mhz a bude ho možné neskôr zreťaziť s inicializačným intervalom `II=1`. Komponent bude mať na vstupe RGB snímku reprezentovanú pomocou dátového typu `float`.

Bude využitá podobná architektúra ako v predchádzajúcom prípade, ale rozdielom je, že celý operátor je rozdelený do troch stupňov. Tento spôsob umožní zapuzdriť logické časti do jednotlivých subkomponentov, ktoré je následne možné použiť aj samostatne, čo uľahčí ladenie (je možné vysyntetizovať jeden a otestovať samostatne). Konkrétne tieto tri subkomponenty sú:

- Predspracovanie (funkcia `TMO_Reinhard05Preprocess`) – extrakcia jasovej zložky
- Mapovanie tónov a výpočet parametrov snímky (funkcia `TMO_Reinhard05ToneMap`)
- Normalizácia do intervalu $< 0, 1 >$ (funkcia `TMO_Reinhard05Postprocess`)

Aby tieto stupne mohli pracovať paralelne, je potrebné v top-level funkcií použiť pragmu: `pragma HLS DATAFLOW`, ktorá umožňuje vytvoriť komponent, ktorý sa skladá z logických častí pracujúcich paralelne. Tieto funkcie si postupne predávajú spracovávané pixely pomocou interných streamov.

Predspracovanie je prvým stupňom a samostatným subkomponentom v rámci dizajnu. Na vstupe má video stream, z ktorého extrahuje jasovú zložku. Spôsob prevodu je obdobný ako pri Dragovom operátore, ale je extrahovaný iba jas pixelu. Následne je vytvorená hodnota pre interný stream, ktorý obsahuje jednotlivé intenzity RGB a intenzitu jasu. V jednej hodnote sú intenzity zložiek reprezentované nasledovne:

- $< 127, 96 >$ – jasová intenzita
- $< 95, 64 >$ – intenzita kanálu R
- $< 63, 32 >$ – intenzita kanálu G
- $< 31, 0 >$ – intenzita kanálu B

Mapovanie tónov prebieha rovnako ako v prípade originálneho algoritmu. Rozdiel je v tom, že je rozbalená najvnútornejšia slučka, ktorá realizuje mapovanie pre každý kanál jedným priechodom cyklu. Tento stupeň zároveň počíta potrebné globálne parametre snímky.

Normalizácia tónov je posledným krokom, kde je každý pixel normalizovaný do rozsahu $< 0, 1 >$. Nakoniec je pixel zapísaný do výstupného streamu.

5.3.2 Výsledky syntézy

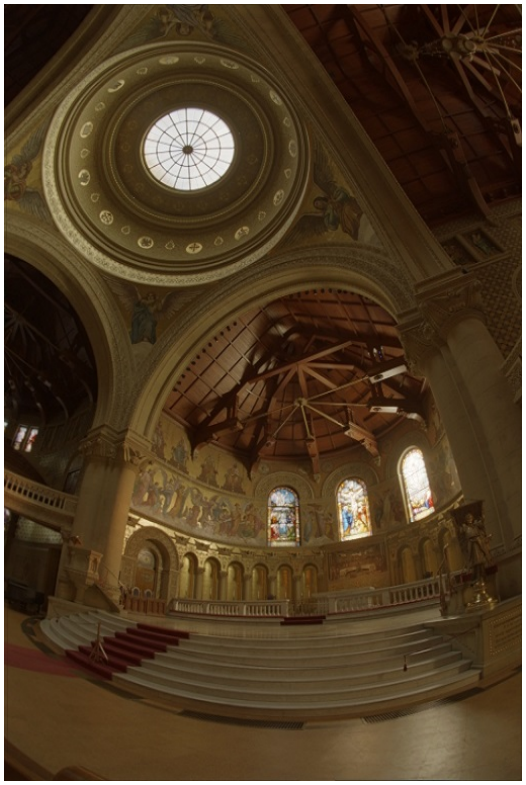
V tabuľke 5.2 sú zobrazené výsledky syntézy pre rôzne pracovné frekvencie, dizajn nie je zreťazený. Operátor je zložitejší ako Dragov operátor a je vidieť veľké zvýšenie nárokov na zdroje. Navýšenie je spôsobne množstvom operácií, ktoré sú pri mapovaní použité.

Rýchlosť [MHz]	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
20	0	125	9850	26659
50	0	118	11813	25392
100	0	118	15803	24963
200	0	118	23768	26590

Tabuľka 5.2: V tabuľke je zobrazený výsledok syntézy s rôznymi pracovnými frekvenciami a bez zreťazenia (ide o odhad využitia po procese HLS).

5.3.3 Ukážky

Nasledujú ukážky výstupov softvérovej implementácie a realizácií v FPGA (obrázky 5.3a a 5.3b), na ktorých je vidieť, že aj Reinhardov operátor správne mapuje intenzity jasu a všetky detaily sú viditeľné. Problémom je väčšia spotreba zdrojov a občas náročnejšie vhodné určenie parameterov pre uspokojivý výsledok.



(a) `memorial.exr` po aplikovaní Reihardovho operátora. Parametre `f=1.0 ,a=1.0, c=1.0`

(b) `designCenter.exr` po aplikovaní Reihardovho operátora. Parametre `f=6, a=0.5, c=0.75`

Obr. 5.3: Ukážka tónového mapovania pomocou Reinhardovho operátora.

5.4 Lokálny operátor Durand a Dorsey

Posledným implementovaným je lokálny operátor, na ktorom je najzložitejšia implementácia bilaterálneho filtra, ktorý pracuje nad jasovými intenzitami obrazu. Vytvorený dizajn vychádza okrem oficiálnej implementácie aj z implementácie dostupnej v knižnici **pfstmo**. SW implementácia je na priloženom CD v zložke `Sources/SW/durand02`.

5.4.1 Implementácia vo Vivado HLS

Táto implementácia sa od predošlých dvoch líši v tom, že namiesto využitia dátového typu `float` sú použité typy s pevnou rádovou čiarkou – `ap_fixed`. Cieľ je rovnaký ako v prípade dvoch predchádzajúcich– vytvoriť komponent pracujúci na frekvencií 100 Mhz a bude ho neskôr možné zreťaziť s inicializačným intervalom `II=1`. Taktiež komponent predpokladá, že na vstupe je snímka reprezentovaná pomocou hodnôt v plávajúcej rádovej čiarke a v HD rozlíšení, tj. 1280×720 pixelov.

Architektúra komponentu je zhodná s architektúrou komponentu Reinhardovho operátora (rozdelenie na stupne a režim `dataflow`). Rozhranie komponentu tvorí Axi Video Stream s dátovou šírkou 96bitov (3×32) a výstup je tiež Axi Video Stream so zhodnou šírkou. Jediný rozdiel je v tom, že po stupni predspracovania sú intenzity vstupných pixelov uložené v dočasnej FIFO fronte, ktorá je priradená internému streamu. Z tejto fronty sú vyberané v poslednom stupni a sú spárované s odpovedajúcou hodnotou jasu. Rozdelenie stupňov je na predspracovanie, bilaterálne filtrovanie a kompresiu intenzít.

Predspracovanie

Predspracovanie je veľmi podobné predchádzajúcemu operátoru. Iné je odosielanie dát do interných streamov. Extrahované hodnoty jasu sú odosielané do subkomponentu bilaterálneho filtru a hodnoty intenzít kanálov RGB sú odosielané priamo do stupňa kompresie.

Bilaterálne filtrovanie

Ide o najzložitejší krok v rámci implementácie tohoto operátoru vo Vivado HLS, pretože vyžaduje využitie riadkových vyrovnávacích pamätí a tzv. okna, nad ktorým sa pracuje. Vstupom do tohto subkomponentu sú intenzity spočítané v prechádzajúcom kroku a výstupom je spracovaná snímka pomocou bilaterálneho filtru. V typickej implementácií v jazyku C/C++ je bilaterálne filtrovanie reprezentované dvoma vnorenými slučkami, ktoré reprezentujú priechod obrazom a ďalšiu vnorenú slučku, ktorá reprezentuje priechod cez konvolučné jadro.

V prípade implementácie pre FPGA je dôležité dodržiavať viacero zásad aby sa dal výsledný dizajn zreťaziť a aby nevyžadoval príliš veľké množstvo zdrojov. Úzkym hrdlom sa často stáva prístup do pamäte, kde sú často využívané BRAM, ktoré sú dvoj-portové a teda nie je fyzický realizovateľné čítanie všetkých hodnôt v jednom takte. Ako už bolo spomenuté, nie je možné uložiť celú snímku v HD rozlíšení. Je však možné načítať časť spracovávanej snímky do BRAM pamätí, ktoré umožnia vytvorenie priechodu cez vstupnú snímku tak, že nebude presiahnutý počet operácií s BRAM pamäťami v jednom takte.

Dôležitú abstrakciu nad týmito pamäťami umožňuje priamo Vivado HLS v knižnici `hls_video.h`. Ide o šablónový objekt `hls::LineBuffer`, ktorý reprezentuje riadkovú vyrovnávaciu FIFO pamäť pre snímku. Dajú sa postupne načítavať pixely vstupnej snímky a následne k nim pristupovať. Poskytuje základné operácie, ako vloženie hodnoty do pamäte

alebo prečítanie. Deklarácia je nasledovná: `hls::LineBuffer< r, c, type > lineB;` Kde premenná `r` reprezentuje počet riadkov, ktoré má táto vyrovnávacia pamäť. Premenná `c` je počet prvkov v jednom riadku. Okrem vloženia a prečítania prvku poskytuje pokročilú funkcionalitu:

- `shift_up(int col)` – posunie stĺpec s indexom `col` hore
- `shift_down(int col)` – posunie stĺpec s indexom `col` dole
- `insert_bottom(T value, int col)` – vloží hodnotu `value` do spodného riadka stĺpca s indexom `col`
- `insert_top(T value, int col)` – vloží hodnotu `value` do vrchného riadka stĺpca s indexom `col`

Hlavným prvkom tejto pamäti je dvojrozmerné pole, ktoré má v tomto prípade šírku 1280 prvkov a výšku 7 prvkov. Podstatné však je to, ako ho Vivado následne vysyntetizuje. Šablóna tohoto typu obsahuje tri direktívy, ktoré sú zobrazené na algoritme 5.1. Prvá direktíva `array_partition` určí Vivadu, aby jednotlivé riadky rozdelil do rozdielnych BRAM pamätí, čo umožní paralelný prístup ku všetkým riadkom. Druhá direktíva `HLS dependence variable=val inter false` je nastavenie falošnej závislosti medzi jednotlivými riadkami. Obdobne je nastavená falošná závislosť aj v rámci jedného riadka.

```
#pragma HLS array_partition variable=val dim=1 complete
#pragma HLS dependence variable=val inter false
#pragma HLS dependence variable=val intra false
```

Algoritmus 5.1: Direktívy, ktoré obsahuje šablóna pre vytvorenie typu hls::LineBuffer.

Teraz je možné realizovať priechod cez obraz s uložením v riadkovej vyrovnávacej pamäti nasledovne:

```
for( uint32_t i = 0; i < TMO_WIDTH * TMO_HEIGHT; i++ )
{
  TMO_StreamInternal pixInImg = intIn.read( ); // precitanie vstupu
  TMO_FixedType I = pixInImg;
  lineBuff.shift_up(coll);          // posunutie stlpca
  lineBuff.insert_top( I, coll ); // vlozenie hodnoty
  ...                               // pokracovanie algoritmu
}
```

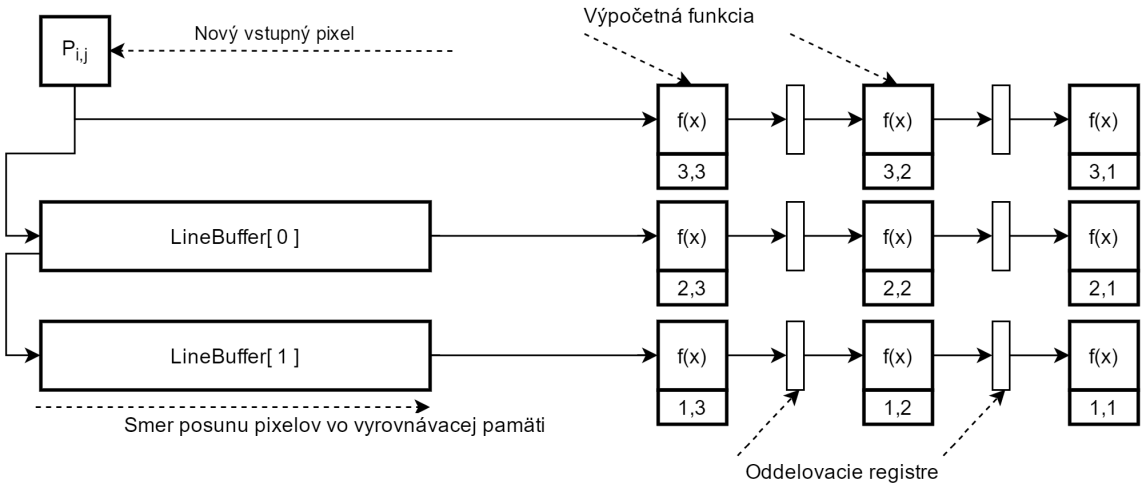
Algoritmus 5.2: Direktívy, ktoré obsahuje šablóna pre vytvorenie typu hls::LineBuffer.

Stále nie je možné ľubovoľne pristupovať k hodnotám v rámci riadkov. Pre takúto možnosť prístupu je potrebné využiť ďalší šablónový typ – `hls::Window`. Ide o dvojrozmerné pole, ku ktorému je možné pristupovať ľubovoľne a je možné prečítať všetky hodnoty v jednom takte. Šablóna tohoto objektu obsahuje nasledovné pragma direktívy:

```
#pragma HLS ARRAY_PARTITION variable=val dim=1 complete
#pragma HLS ARRAY_PARTITION variable=val dim=2 complete
```

Algoritmus 5.3: Direktívy, ktoré obsahuje šablóna pre vytvorenie typu hls::LineBuffer.

Tieto direktívy určia Vivadu aby dvojdimenzionálne pole rozdelil do rôznych pamätí. Takéto okno je realizované pomocou sady nezávislých registrov. Architektúru s využitím riadkovej vyrovnávacej pamäti a okna je možné vidieť na obrázku 5.4 a umožňuje zreťazenie s inicializačným intervalom $II = 1$. Pixely vstupného obrazu sa posielajú postupne po riadkoch zľava doprava a zhora dole. V prípade zreťazenia sa v každom takte nasunie nový pixel do linky na spracovanie a zároveň sa vloží aj do prvého riadku vyrovnávacej pamäte, zvyšné pixely sa získajú zo začiatku riadkových vyrovnávacích pamätí. To umožňuje v každom takte posunúť pixely doprava a vložiť nový stĺpec. Takto je získaná nová oblasť pod konvolučnou maskou v každom takte.



Obr. 5.4: Návrh architektúry pre realizovanie bilaterálneho filtru. Pre zjednodušenie je konvolučné jadro veľkosti 3×3. Skutočná pozícia v rámci konvolučného jadra sa nachádza v rámčeku pod odpovedajúcou funkciou f(x).

Pomocou takéhoto okna je reprezentovaná aktuálne spracovávaná oblasť a postup je nasledovný:

1. do okna je vložený nový pixel prečítaný zo streamu, ktorý sa aj uloží do riadkovej pamäte
2. do okna je vložený nový stĺpec, ktorý je načítaný z riadkovej vyrovnávacej pamäte.

Teraz už je možné pracovať nad oknom, do ktorého sa v každom takte vloží nový stĺpec a takto sa posúva okno postupne zľava doprava a zhora dole. Pre pixel v strede masky je spočítaná nová hodnota, ktorá je závislá na vzdialenosti pixelu od stredu masky a váhy rozdielu ich intenzít. Pre implementáciu sa ukázala ako vhodná šírka konvolučného jadra hodnota 7×7 (vzhľadom na dostupné LUT v FPGA). Ukážka výsledku bilaterálneho filtrovania je na obrázku B.1 v prílohe B, kde po bilaterálnom filtrovaní dochádza k rozostreniu, ale hrany zostávajú zachované.

Kompresia intenzít

Kompresia intenzít je už jednoduchý krok, v ktorom sú intenzity postupne čítané z interných streamov (jeden obsahuje hodnoty RGB a druhý hodnoty z bilaterálneho filtru). Po načítaní sú pomocou výpočtov, ktoré sú uvedené v kapitole 3.5 prepočítané na nové hodnoty. Nakoniec sú zapísané do výstupného video streamu a sú v rozsahu $< 0, 1 >$.

5.4.2 Výsledky syntézy

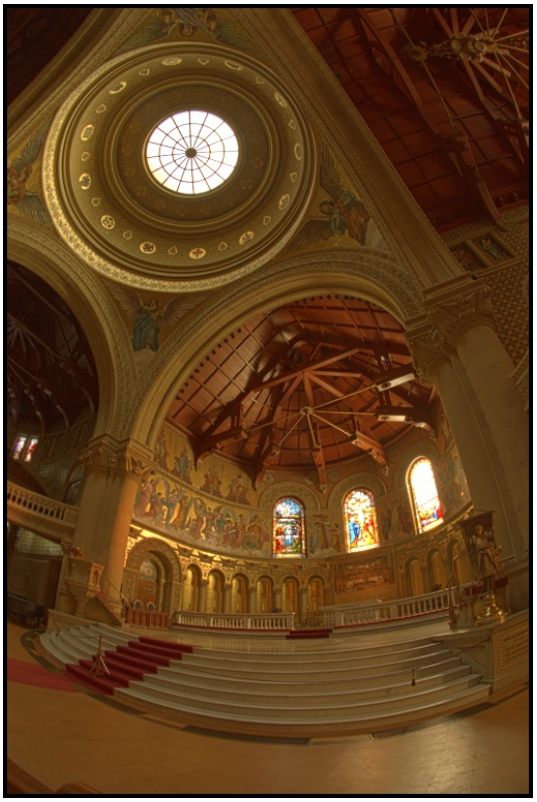
V tabuľke 5.3 sú zobrazené výsledky syntézy, kde ako pracovná frekvencia je 100 Mhz a dizajn nie je zreťazený. Najviac zdrojov je spotrebovaných na vytvorenie subkomponentu bilaterálneho filtra, ktorý obsahuje vo veľkej miere násobičky.

Rýchlosť [MHz]	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
20	51	71	5995	24474
50	51	71	8008	24493
100	51	71	11773	25006
200	51	71	21883	29314

Tabuľka 5.3: V tabuľke je výsledok syntézy pre rôzne pracovné frekvencie bez reťazenia.

5.4.3 Ukážky

Na obrázkoch 5.5a a 5.5b sa nachádzajú ukážky výstupov po mapovaní tónov. Na obrázkoch je vidieť, že operátor produkuje vizuálne dobrý výsledok, avšak bolo by vhodné použiť bilaterálny filter s väčšou maskou.



(a) Snímka `memorial.exr` po aplikovaní Durandovho operátora.

(b) Snímka `designCenter.exr` po aplikovaní Durandovho operátora.

Obr. 5.5: Na obrázku 5.5a a 5.5b sú výsledky po aplikovaní operátora. Parametre gaussovej funkcie $\sigma_s = 1.5, \sigma_r = 0.75$.

5.5 Optimalizácie operátorov

V tejto podkapitole je pozornosť venovaná základným optimalizáciam výpočtu, ktoré boli využité. Najskôr je ukázaný spôsob ako zreťaziť výpočet s inicializačným intervalom `II=1`. Následne je ukázaný vplyv zreťazenia na zdroje.

Jednou zo základných techník ako urýchliť výpočet je využitie reťazenia v rámci slučky. V prípade Vivado HLS je zreťazenie realizované pomocou direktívy `HLS PIPELINE II=x` (alebo obdobne v .tlc súbore), kde x vyjadruje inicializačný interval, tj. počet taktov, po ktorých sa začne nový výpočet. Cieľom zreťazenia je hlavne dosiahnutie toho, aby výpočet novej iterácie cyklu začal v každom hodinovom takte. Často však nastane problém s prístupom k premenným a k explózii využitých zdrojov. Hlavným zdrojom problémov pri premenných sú premenné zdieľané medzi iteráciami cyklov. V prípade zdrojov sú to hlavne zložitejšie výpočty, ako napríklad rôzne logaritmické a exponenciálne funkcie.

5.5.1 Dragov a Reinhardov operátor

V prípade Dragovho aj Reinhardovho operátoru sa pri pokuse o zreťazenie Vivado pokúsi dizajn reťaziť, ale narazí na problém s premennou `lumSumNew`, ktorá reprezentuje logaritmickú sumu jasových intenzít. Pri Dragovom operátore ide o výpočet výrazu `lumSumNew += logf( 2.3e-5f + MAX( 0.0, chanY ) );`, ktorý prebieha v každej iterácií. Vivado vyhlási upozornenie, ktoré hovorí, že nie je možné zreťaziť dizajn s `II=1`, pretože výpočet obsahuje závislosť. Nástroj vo výsledku dizajn zreťazí, ale s `II=5`. Problémom je, že k premennej `lumSumNew`, z ktorej sa počíta suma, je pripočítaná hodnota. Toto je spôsobené operáciou sčítania v plávajúcej čiarke, ktorá trvá 5 taktov a nazýva sa konflikt typu RAW (Read After Write). Teda aktuálnu hodnotu `lumSumNew` je možné použiť až po 5 taktoch, ale pre zreťazenie je vyžadované, aby načítanie, výpočet a sčítanie tejto hodnoty bolo hotové v jednom takte. Toto miesto sa stáva úzkym hrdlom výpočtu.

Riešením je vytvorenie malého poľa, do ktorého sa budú postupne ukladať čiastkové súčty, ktoré stačí po ukončení hlavnej slučky sčítať. Tento princíp je zobrazený na nasledujúcom algoritme 5.4. Veľkosť tohoto poľa stačí 5 prvkov, keďže samotné sčítanie trvá 5 taktov. Aby bolo zaistené, že sa nepristupuje k rovnakému prvku poľa, je použitý operátor modulo, ktorým sa cyklicky pristupuje k jednotlivým pozíciám.

```
1 float partialLogSum[ PARTIAL_SUM_SIZE ];
2
3 for( uint32_t i = 0; i < TMO_WIDTH * TMO_HEIGHT; i++ )
4 {
5 #pragma HLS pipeline II=1
6   ...
7   partialLogSum[ i % PARTIAL_SUM_SIZE ] += logf( 2.3e-5f + chanY );
8   ...
```

Algoritmus 5.4: Varovanie, ktoré vypíše Vivado HLS pri pokuse o zreťazenie základnej implementácie.

Po tejto úprave už je zreťazenie s `II=1` realizovateľné, na čo upozorní aj Vivado: `Pipelining result: Target II: 1, Final II: 1, Depth: 46`.

5.5.2 Durandov operátor

Durandov operátor je najzložitejším operátorom a vyžaduje väčšie množstvo optimalizácií pre efektívnu realizáciu v FPGA. Základnou optimalizáciou je využitie typu `ap_fixed`. Problémom však je samotná podstata HDR snímok, ktoré môžu dosahovať rozsiahle intervaly hodnôt. Preto je na začiatku pri extrakcií jasového kanálu použitý široký dátový typ o veľkosti 42 bitov (znamienkový, 18 bitov celá časť). V ďalšom kroku je jas logaritmovaný a teda pre jeho reprezentáciu stačí dátový typ, ktorý má menšie rozlíšenie pre celú časť, konkrétne o šírke 18 bitov (znamienkový, 6 bitov celá časť). Tento typ je propagovaný cez celý dizajn až do procesu kompresie intenzít.

Pri výpočte hodnôt bilaterálneho filtru je využitá vlastnosť Vivada, ktoré pri reťazení automaticky rozbaľuje všetky slučky a extrahuje konštanty v rámci jednotlivých iterácií. Na algoritme 5.5 je ukázaný spôsob zápisu výrazu, ktorý vyhodnotí Vivado HLS ako konštantu, ktorá je jedinečná pre každú iteráciu. Táto konštanta je následne implementovaná ako ROM register.

```
1  for ( int8_t kerI = ( -KER_SIZE / 2); kerI < KER_SIZE / 2 ; kerI++ )
2  {
3    for ( int8_t kerJ = ( -KER_SIZE / 2); kerJ < KER_SIZE / 2 ; kerJ++ )
4    {
5      const float kernelValue = -( kerI*kerI + kerJ*kerJ) / 2.0*sigS*sigS;
6      const TMO_FixedNormType gs = expf( kernelValue );
7      ...
8    }
9    ...
10 }
```

Algoritmus 5.5: Zápis výpočtu, ktorý Vivado vyhodnotí ako konštantu.

Ostatné optimalizácie sú zhodné ako u predchádzajúcich operátorov a zahŕňajú prevody výrazov a extrakcie konštánt. Výslednú implementáciu operátoru je možné zreťaziť s II=1 a maximálnou pracovnou frekvenciou 125Mhz. Najväčší vplyv na spotrebu zdrojov má samozrejme veľkosť bilaterálneho filtru. V tabuľke 5.4 sa nachádzajú výsledky syntézy pre rôzne veľké filtrovacie jadrá.

Veľkosť kernelu	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
3x3	42	63	11670	23699
5x5	52	87	14120	28443
7x7	66	122	18416	38386
9x9	83	255	24059	52096

Tabuľka 5.4: Tabuľka obsahuje odhad spotreby zdrojov pre Durandov operátor. Dizajn je zreťazený s $II = 1$ a pracovnou frekvenciou 100Mhz.

5.5.3 Výsledná spotreba zdrojov

V tabuľke 5.5 sa nachádzajú odhady spotreby zdrojov. Najviac vyžaduje Durandov operátor, čo je spôsobené jeho komplexnosťou.

Operátor	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
Drago03	4	176	22854	28437
Reinhard05	18	196	21185	34454
Durand02 (7×7)	66	122	18416	38386

Tabuľka 5.5: Výsledné počty zdrojov, ktoré zaberajú komponenty. Dizajny sú zreťazené s $II = 1$ a pracovnou frekvenciou 100Mhz.

5.5.4 Výsledky

Pre vyhodnotenie implementácie operátorov v FPGA sú porovnané dve hľadiská. Prvým je porovnanie výsledných snímok po mapovaní tónov. Takto je možné overiť správnosť implementácie (ak je považovaná SW implementácia za správnu). Po tomto porovnaní je vhodné porovnať aj dosiahnuté zrýchlenie, aby vo výsledku bola overená vhodnosť implementácie. Pre porovnanie sú zvolené tri platformy. Prvou je samotný ARM A9 procesor, ktorý je dostupný na Xilinx Zynq, jeho pracovná frekvencia je 866 MHz. Druhou platformou je FPGA na Xilinx Zynq, kde komponenty pracujú na frekvencií 100 MHz a sú zreťazené s $II = 1$. Ako posledným je procesor Intel Core I5 6600 Skylake so základnou pracovnou frekvenciou 3.3 GHz. Testovacie snímky sú upravené na veľkosť 1280×720 a sú vo formáte `.exr`.

V tabuľke 5.6 sa nachádzajú hodnoty porovnania FPGA implementácií voči SW, ktorá beží na ARM procesore. Porovnané sú výsledné snímky vo formáte 8bit RGB a je zobrazená priemerná chyba pre každý kanál samostatne. Hodnota 0 by znamenala úplnú presnosť a čím je vyššia, tým je väčšia odchýlka. Porovnanie výsledných snímok je dôležité hlavne pri Durandovom operátore, pretože využíva aritmetiku v pevnej rádovej čiarke a to vnáša do výpočtu určitú odchýlku. Pri Dragovom a Reinhardovom je použitý výpočet v pohyblivej rádovej čiarke a pri testoch sa ukázalo, že chyba sa pohybuje na úrovni stotín až tisícin, rozdiel intenzity je maximálne 1, čo je spôsobené zaokrúhlením výsledku. Pri Durandovom operátore sa pohybuje v rádoch jednotiek a vzhľadom k dosiahnutému zrýchleniu ide o prijateľnú chybu a výsledné snímky sú na „pohľad“ rovnaké. Pre spresnenie výpočtu by bolo potrebné rozšíriť dátové typy s pevnou rádovou čiarkou o ďalšie bity, čo by viedlo na väčšiu spotrebu zdrojov.

Metóda	**Drago03**			**Reinhard05**			**Durand02** (7×7)		
Snímka/Kanál	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**
memorial	0.044	0.021	0.052	0.0002	0.0003	0.0002	1.421	0.753	0.362
designCenter	0.172	0.010	0.309	0	0	0	1.034	1.037	0.904
Ocean	0.003	0.000	0.000	0	0	0	1.145	1.341	1.263
MtTamWest	0.005	0.001	0.012	0	0	0	0.963	1.157	0.998

Tabuľka 5.6: V tabuľke je zobrazená priemerná chyba na kanál pre rôzne snímky. V prípade Durand02 je vidieť, že použitie pevnej rádovej čiarky vnáša do výpočtu určitú chybu.

V tabuľke 5.7 sú zobrazené porovnania rýchlostí spracovania operátorov na CPU a FPGA. Každé meranie bolo realizované 10× a vypočítaný priemer. Vstupná snímka je zhodná pre všetky porovnávané architektúry a čas merania zahŕňa iba samotný výpočet mapovania. FPGA komponenty bežia na **100Mhz** a sú zreťazené s **II=1**. Pri SW implementácií sú pri preklade použité optimalizácie na úrovni **-O3**.

Metóda/Platforma	**Intel Core I5**	**Arm A9**	**FPGA**
Drago03	116ms	1526ms	22ms
Reinhard05	111ms	2640ms	67ms
Durand02 (7×7)	481ms	3147ms	22ms

Tabuľka 5.7: Tabuľka obsahuje namerané časy pre operátory na rôznych platformách.

V tabuľke 5.8 sú dosiahnuté priemerné časy prepočítané na zrýchlenie. V prípade Intel Core I5 nie je FPGA až tak razantne rýchlejšie, ale v čom vyniká FPGA implementácia, je spotreba. Pri porovnaní zrýchlenia oproti ARM procesoru je vidieť rádové zrýchlenie. V konečnom dôsledku implementácia v FPGA poráža ako ARM procesor, tak aj moderný procesor Intel Core I5. Pri FPGA implementácií sú však použité dva priechody aby bol výsledný snímok presný (v prvom volaní sa spočítajú príznaky snímky a v druhom sa namapuje snímok s vypočítanými hodnotami). Ak by sa použila technika použitia predchádzajúcich globálnych príznakov pre výpočet aktuálnej snímky, bola by dosiahnutá vyššia rýchlosť a snímková frekvencia. Pri Reinhardovom operátore je pomalšia doba behu spôsobená nutnosťou viacerých priechodov vstupným obrazom.

Metóda/Platforma	**Intel Core I5**	**Arm A9**
Drago03	5.2x	69.4x
Reinhard05	1.6x	39.4x
Durand02 (7×7)	21.9x	143x

Tabuľka 5.8: Tabuľka obsahuje dosiahnuté zrýchlenie FPGA oproti ostatným platformám.

5.5.5 Výhody a nevýhody Vivado HLS

Nespornou výhodou je už samotná možnosť zápisu algoritmov jazyku C/C++ alebo SystemC, ale určite je najlepšie využívať C++, pretože narozdiel od C poskytuje dátové typy s vlastnou šírkou. Plusom je aj rýchle pochopenie princípov syntézy na systémovej úrovni. Je však potrebné sa zbaviť návyku z imperatívnych programovacích jazykov, kde jednotlivé príkazy postupujú sekvenčne za sebou. Treba kód chápať ako množstvo paralelných blokov (cykly, postupnosti výrazov), ktoré sa plánujú čo najskôr to závislostí medzi nimi umožnia.

Hlavná nevýhoda je absencia plnej kontroly nad výsledným dizajnom. Nie je možné do detailu naplánovať operácie a ostáva iba spoľahnutie na syntézny nástroj. Takisto sa negeneruje úplne optimálny kód a často sú implementácie v HDL jazykoch optimálnejšie a s menšou spotrebou zdrojov. Často sa najlepším zdrojom informácií ukázali samotné hlavičkové súbory `hls_xxxx.h`, ktoré často obsahujú implementáciu pre ladenie a dá sa z nich veľa pochopiť. Nakoniec je dobré odporučiť, aby sa návrhári držali návrhov dizajnov, ktoré sú obsahom príkladov, pretože občas Vivado chybne interpretuje návrhárov zámer.

5.6 Realizácia aplikácie na Xilinx Zynq

Posledným krokom je realizácia finálnej aplikácie. Najskôr je potrebné vytvoriť dizajn v prostredí Vivado a po procese syntézy sa dá presunúť k implementácií finálnej aplikácie.

5.6.1 Dizajn v prostredí Vivado

Pre vytvorenie finálneho dizajnu je potrebné vytvoriť schému aplikácie v prostredí Vivado. V tomto prostredí je možné jednoducho vytvoriť schému pomocou editoru, ktorý umožňuje jednoducho pridávať komponenty do schémy a prepájať ich. Pridanie novej komponenty do schémy je z časti automatické, kde pomocou sprievodcu je možné automaticky pripojiť rozvod hodín, signál reset a riadiace signály. Ostatné prepojenia je potrebné vytvoriť ručne, ale nie je to problém, pretože Vivado automaticky zoskupuje signály protokolu do jedného portu. Potom stačí jednoducho spojiť požadované porty a Vivado automaticky prepojí odpovedajúce signály. Finálna schéma sa nachádza v prílohe C.

Vivado automaticky ku komponentom generuje aj ich fyzické adresy, na ktorých sú mapované jednotlivé komponenty a ich konfiguračné registre, ktoré sú prenášané pomocou AXI4-Lite protokolu. V prípade potreby je možné realizovať editáciu v karte „Address editor“. Keďže XIlinx Zynq je postavený na 32 bitovej architektúre, dá sa maximálne adresovať 4GB pamäte. Samotné FPGA komponenty sú mapované od adresy 0x4000000. Následne je potrebné vykonať proces syntézy a vygenerovať výsledný bitový súbor, ktorý sa uloží do pamäte a nahrá sa do FPGA pri štarte systému Zynq.

5.6.2 Aplikácia na Xilinx Zynq

Na kite Zynq beží OpenSuse vo verzií 13.1 a je možné využívať všetky výhody operačného systému. Takýmto príkladom je testovacia aplikácia, ktorá bola použitá na overovanie implementovaných komponentov pomocou Vivado HLS. Táto aplikácia využíva `X11 forwarding` pre prenos výsledkov tónového mapovania na hostiteľský počítač pomocou SSH pripojenia.

Menším problémom je prístup k adresám FPGA komponentov, keďže bežiace aplikácie využívajú virtuálne adresy. Pre prístup k fyzickej adrese je potrebné využiť funkciu `mmap`, ktorá mapuje súbory a zariadenia do virtuálnej pamäte. Tak isto je potrebné vytvoriť určité miesto v pamäti, do ktorých sa budú ukladať snímky pre VDMA komponent. VDMA pracuje nad fyzickými adresami a preto je potrebné vytvoriť takú oblasť v pamäti, ktorú jadro operačného systému nebude prideľovať bežiacim aplikáciam. Pre tento účel sa dá využiť DeviceTree. DeviceTree je dátová štruktúra popisujúca hardware, ktorým disponuje hostiteľský systém. V tejto štruktúre je možné popisovať aj jednotlivé periférie ako I2C, SPI a podobne. Takisto sa dá určiť rozsah pamäte, ktorú bude využívať operačný systém a to pomocou parametra `MEM=xxxMB`. V tomto prípade bude operačný systém využívať 768MB pamäte a 256 bude vyhradených pre uloženie prijatých snímok z kamery. Samotný pamäťový priestor je teda rozdelný na:

- `0000 0000h - 2FFF FFFFh` (768MB) – pamäť RAM využitá linuxovým jadrom
- `3000 0000h - 3FFF FFFFh` (256MB) – pamäť pre snímky
- `4000 0000h - FFFF FFFFh` (3GB) – ostatné komponenty - M_AXI_GP0, M_AXI_GP1, periférie

5.6.3 Prepojenie komponentov v FPGA so SW aplikáciou

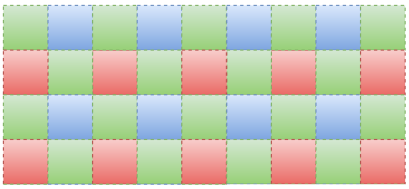
Vivado priamo generuje ovládače pre všetky komponenty, ktoré sa nachádzajú v dizajne a poskytuje export do prostredia Xilinx SDK. Toto prostredie je určené pre softwarovú implementáciu aplikácií a priamo podporuje tieto ovládače. Avšak takýto ovládač je veľmi jednoduchý, takže pre riadenie FPGA komponent sú využité skripty. Stačí na adresu, ktorá je predom definovaná vo Vivade, zapísať štartovací bit a komponenta začne pracovať.

5.6.4 Výsledná aplikácia

Výsledná aplikácia je riadená z príkazového riadka a po spustení a nakonfigurovaní, začne zachytávať dáta z pripojenej kamery. Tieto dáta sú predané pomocou VDMA do tónovacej komponenty, kde prebehne mapovanie tónov a odoslanie späť do pamäte. Tu prevezme dáta druhá VDMA, ktorá ich posiela na spracovanie pre HDMI výstup. Najskôr sú konvertované do farebného formátu YUV 4:4:4 a potom prevzorkované do formátu YUV 4:2:2.

5.6.5 Priebeh spracovania

Pre prijímanie vstupných dát z kamery je využitá knižnica pre ovládanie multi-expozičnej kamery, ktorá po upravení využíva tri vlákna. Jedno vlákno prijíma dáta cez UDP protokol, druhé vytvára LDR snímky a predáva ich tretiemu vláknu, ktoré realizuje skladanie snímok. Obsahom prijatých snímok z kamery je tzv. Bayerova maska. Táto reprezentácia je úzko zviazaná s fyzickou realizáciou snímačov v kamerách. Maska obsahuje tri typy filtrov, pričom každý z nich prijíma svetlo inej vlnovej dĺžky (RGB). Usporiadanie je pravidelné a jednoduchú schému je možné vidieť na obrázku 5.6. Výhodou je, že pre každý pixel v obraze je potrebný iba jeden filter, ale farby pixelov je potrebné softwarovo dopočítať. Realizáciu takejto funkcie poskytuje priamo OpenCV, ktorá je využitá.



Obr. 5.6: Ukážka časti Bayerovej masky.

Keďže je potrebné vytvorenie HDR snímku, je využitý multi-expozičný režim kamery, kde jednu HDR snímku tvoria 3 LDR snímky s rôznymi dobami expozície. Tieto časy je možné konfigurovať pomocou konfiguračného súboru. Po prijatí a uložení snímok do pamäte, je vykonané tretím vláknom zloženie do jednej HDR snímky pomocou postupu popísaného v kapitole 3.1.2. Ako váhová funkcia je využitá funkcia Debevec-Malik [13], ktorá minimalizuje váhy prexponovaných a podexponovaných pixelov. Táto snímka je priamo uložená vo vyhradenej časti pamäte.

DMA prenos automaticky zaistí transfer dát do komponenty mapovania tónov, kde je stream postupne spracovaný a predaný späť do pamäte. Keďže naraz pracujú dva DMA prenosy, je potrebné zaistiť aby nedochádzalo ku konfliktom. Obidve DMA cyklicky prechádzajú cez tri zdieľané vyrovnávajúce pamäte, ale aby nedochádzalo ku konfliktom, je využitý režim `GenLock`, ktorý umožňuje nastaviť jednu DMA ako Master a druhú ako Slave. V tomto režime Master generuje pre Slave index, ktorý určuje aktuálnu pamäť, s ktorou pracuje. Tým je zaručené vylúčenie súbežného prístupu.

5.6.6 Výsledná spotreba zdrojov

V nasledujúcej tabuľke 5.9 sa nachádza výsledná spotreba zdrojov aplikácie, ktorá ako tónovací komponent využíva Durandov operátor s veľkosťou konvolučného okna 7×7. Z tabuľky vyplýva, že je využitých takmer 57% LUT a 50% DSP48 jednotiek. Po procese syntézy je skutočná spotreba zdrojov oveľa nižšia oproti odhadu po procese HLS syntézy.

Typ	**Využité**	**Dostupné**	**Využitie v [%]**
FF	31074	106400	29.20
LUT	30279	53200	56.92
Pamäťová LUT	2360	17400	13.56
I/O	20	200	10.00
BRAM	44.5	140	31.79
DSP48	110	220	50.00
BUFG	1	32	3.12

Tabuľka 5.9: Výsledná spotreba zdrojov aplikácie po procese syntézy.

5.6.7 Výsledky

Na nasledujúcej sekvencií štyroch obrázkov je ukážka výstupu aplikácie. Táto sekvencia bola zaznamenaná v silne osvetlenej miestnosti. Na výslednom obrázku 5.7d je vidieť detaily vo všetkých častiach scény. Oproti 5.7a sú viditeľné detaily vo všetkých miestach a v porovnaní s 5.7c sú potlačené odlesky. Ďalšie ukážky sa nachádzajú na priloženom CD.

(a) LDR (4000us expozičný čas)

(b) LDR (16000us expozičný čas)

(c) LDR (64000us expozičný čas)

(d) Po mapovaní.

Obr. 5.7: Ukážka tónového mapovania pomocou Durandovho operátoru. Na snímkach 5.7a, 5.7b , 5.7c sú LDR snímky, z ktorých bol zložený HDR snímok a namapovaný pomocou Durandovho operátoru, ktorý je na 5.7d.

Kapitola 6

Záver

V diplomovej práci som sa zameral na problematiku mapovania tónov s využitím syntézy na systémovej úrovni a zadanie sa podarilo splniť úplne. Prvým krokom bolo získanie teoretických znalostí o syntéznom nástroji Vivado HLS a platforme Xilinx Zynq. Tieto poznatky sú obsahom kapitoly 2, ktoré sa ukázali ako veľmi dôležité pre praktické využitie syntézneho nástroja Vivado HLS. Druhý bod zadania, preštudovania problematiky obrazu s vysokým dynamickým rozsahom a mapovania tónov, je obsahom kapitoly 3. Tieto znalosti mi poskytli dobrý základ pre následnú implementáciu aplikácie. Nakoniec kapitola 5 obsahuje popis implementácie zvolených operátorov v prostredí Vivado HLS. Jednotlivé operátory boli zoptimalizované na veľkú priepustnosť a sú schopné spracovania HDR obrazu v HD rozlíšení v reálnom čase. Ďalej sú operátory porovnané z hľadiska presnosti a výkonnosti oproti iným platformám. Posledným krokom je otestovanie v reálnej aplikácií, ktorá beží na platforme Xilinx Zynq.

Zvolená problematika mapovania tónov sa ukázala ako vhodnou na overenie vlastností Vivado HLS, keďže implementované metódy využívajú široké spektrum funkcií, ktoré sú prostredím poskytnuté. Dosiahnuté zrýchlenie ukazuje, že syntéza z vyšších programovacích jazykov je veľmi perspektívna oblasť, keďže jej využitie poskytuje veľmi dobrú abstrakciu od samotného prepojenia v FPGA. Nevýhodou je, že návrhár stále musí dodržiavať určité pravidlá pre implementáciu, ale akonáhle ich má zvládnuté, je práca s Vivado HLS rýchla a priamočiara.

Ako rozšírenia práce je možné realizovať ďalšie typy operátorov alebo preskúmať možnosti realizácie operátoru Durand a Dorsey pomocou Rýchlej Fourierovej transformácie. Ďalej je možné urýchliť samotnú tvorbu HDR snímkov v FPGA a vytvoriť tak kompletné akcelerované spracovanie. Bolo by však potrebné využiť platformu s väčšou FPGA. Inou cestou môže byť využitie SIMD akcelerátoru Neon, ktorý je vhodný práve pre takéto typy operácií.

Literatúra

[1] Semiconductor Industry Association (SIA): *International Technology Roadmap for Semiconductors: 1999 edition*. Semiconductor Industry Association (SIA), 1999, 36 s.

[2] Coussy, P.; Gajski, D. D.; Meredith, M.; aj.: An Introduction to High-Level Synthesis. *Design Test of Computers, IEEE*, July 2009: s. 8–17, ISSN 0740-7475.

[3] de Jong, G. G.: Data Flow Graphs: System Specification with the Most Unrestricted Semantics. In *Proceedings of the Conference on European Design Automation*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, ISBN 0-8186-2130-3, s. 401–405.

[4] McFarland, M. C.; Parker, A. C.; Camposano, R.: Tutorial on High-level Synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, ISBN 0-8186-8864-5, s. 330–336.

[5] Tomáš, M.: Pokročilé metody syntézy číslicových obvodů. 2015, predmet Hardware/Software Codesign.

[6] Fingeroff, M.: *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010, ISBN 1450097243, 9781450097246, 35-84 s.

[7] Xilinx: Introduction to High-Level Synthesis with Vivado HLS. [online]. [cit. 2016-05-08].
URL `http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf`

[8] Xilinx: High-Level Synthesis. 2014, [online]. [cit. 2016-05-08].
URL `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug902-vivado-high-level-synthesis.pdf`

[9] Crockett, L. H.; Elliot, R. A.; Enderwitz, M. A.; aj.: *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014, ISBN 099297870X, 9780992978709.

[10] Xilinx: 7 Series FPGAs Configurable Logic Block. 2014, [online]. [cit. 2016-05-08].
URL `http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`

[11] Xilinx: AXI Reference Guide. [online]. [cit. 2016-05-08].
URL `http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf`

[12] Čadík, M.: Evaluation of existing tone mapping operators. [online]. [cit. 2016-05-10]. URL `http://cadik.posvete.cz/tmo/`

[13] Reinhard, E.; Ward, G.; Pattanaik, S.; aj.: *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005, ISBN 0125852630.

[14] Drago, F.; Myszkowski, K.; Annen, T.; aj.: Adaptive logarithmic mapping for displaying high contrast scenes. In *Computer Graphics Forum*, ročník 22, Wiley Online Library, 2003, s. 419–426.

[15] Reinhard, E.; Devlin, K.: Dynamic range reduction inspired by photoreceptor physiology. *IEEE Transactions on Visualization and Computer Graphics*, ročník 11, č. 1, Jan 2005: s. 13–24, ISSN 1077-2626.

[16] Durand, F.; Dorsey, J.: Fast Bilateral Filtering for the Display of High-dynamic-range Images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-521-1, s. 257–266.

[17] Paris, S.; Kornprobst, P.; Tumblin, J.; aj.: *Bilateral Filtering*. Hanover, MA, USA: Now Publishers Inc., 2009, ISBN 160198250X, 9781601982506.

Prílohy

Zoznam príloh

A Obsah CD **68**

B Ukážka bilaterálneho filtrovania **69**

C Schéma aplikácie **70**

Príloha A

Obsah CD

Priložené CD obsahuje:

- Všetky zdrojové kódy
- Dizajn v prostredí Vivado
- Testovacie obrázky
- Ukážky výstupných obrázkov pre jednotlivé operátory

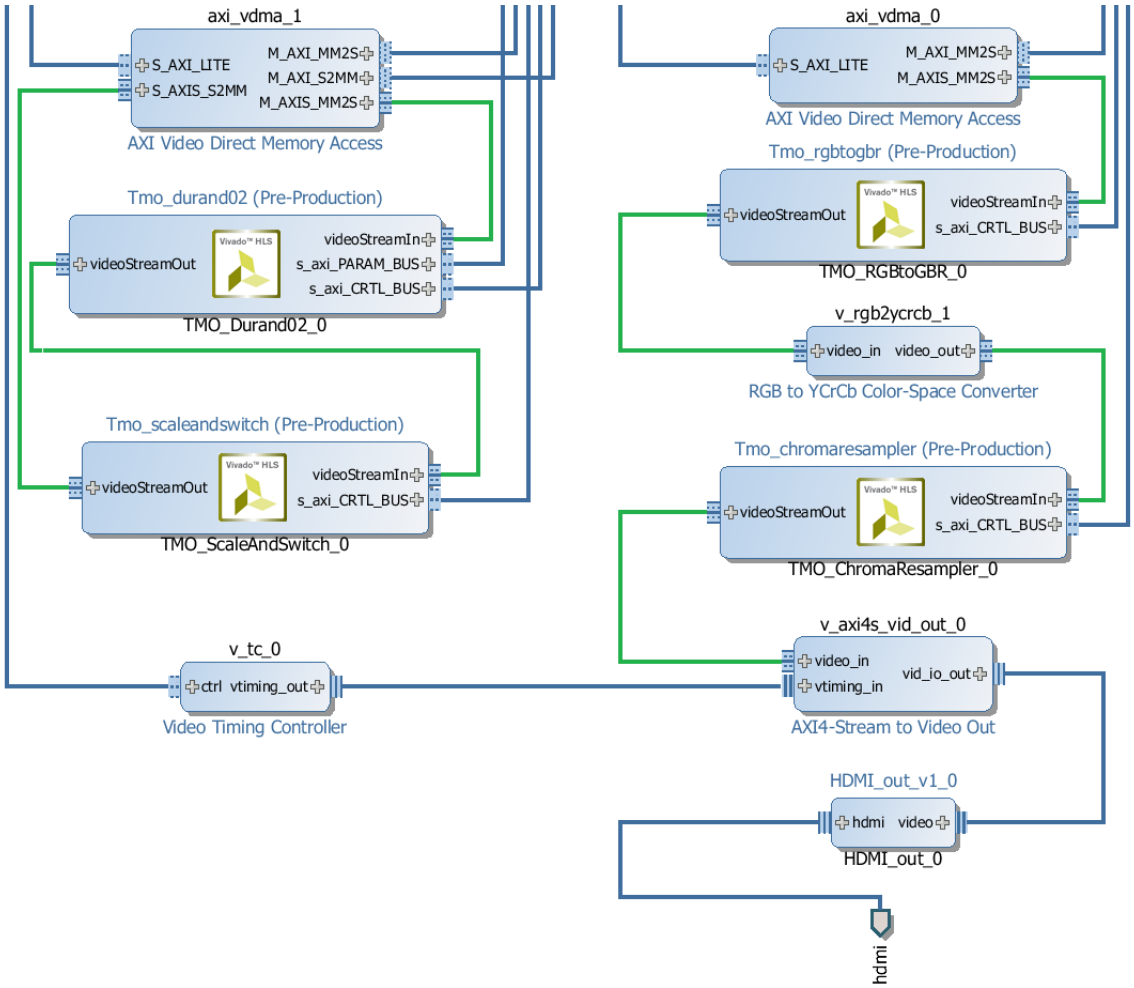
Príloha B

Ukážka bilaterálneho filtrovania

Obr. B.1: Na hornom obrázku sú zobrazené hodnoty intenzít, lineárne namapované do intervalu $< 0, 255 >$. Spodný obrázok je po bilaterálnom filtrovaní. Parametre gaussovej funkcie $\sigma_s = 2, \sigma_r = 0.85$. Najlepšie je výsledky vidieť na detailoch stien, kde v prípade bilaterálneho filtru sú rozmazané a v originálnom obraze sú detaily ostré.

Príloha C

Schéma aplikácie



Obr. C.1: Hlavná časť schémy, ktorá bola vytvorená v prostredí Vivado. Zelenou farbou sú zvýraznené AXI-4 Stream prepojenia.