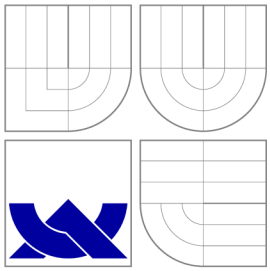
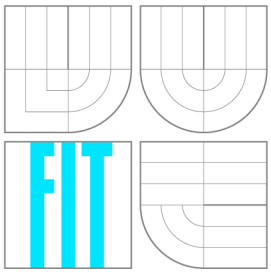


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYUŽITÍ SYNTÉZY NA SYSTÉMOVÉ ÚROVNI PRO APLIKACE S PLATFORMOU ZYNQ

USING HIGH-LEVEL SYNTHESIS FOR ZYNQ PLATFORM APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ HUSÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. OTTO FUČÍK

BRNO 2015

Abstrakt

Práce se zabývá využitím syntézy na systémové úrovni v aplikaci pro zpracování obrazu. Aplikace je určena pro platformu Xilinx ZYNQ. Komponenty v FPGA jsou popsány v jazyce C++. K implementaci bylo použito vývojové prostředí Xilinx Vivado HLS. V rámci práce byly navrženy a implementovány filtry obrazu (Sobelův, mediánový, bilaterální) a také architektura ke klasifikátoru AdaBoost pro detekci registračních značek vozidel. Jako rozšíření byla implementována komponenta pro vyhledávání začátku paketu.

Abstract

This work describes using High-Level Synthesis in image processing application. The application is for Xilinx ZYNQ platform. The source code of components for FPGA is written in C++ programming language. For High-Level Synthesis is used Xilinx Vivado HLS tool. In the application are designed and implemented Sobel filter, Median filter, Bilateral filter and architecture for AdaBoost classificator. The extension of this work is implemented the component for network traffic. The component finds the begin of the packet.

Klíčová slova

Syntéza na systémové úrovni, Xilinx ZYNQ, Vivado Design Suite, FPGA, zpracování obrazu.

Keywords

High-level synthesis, Xilinx ZYNQ, Vivado Design Suite, FPGA, Image Processing.

Citace

Jiří Husák: Využití syntézy na systémové úrovni
pro aplikace s platformou ZYNQ, diplomová práce, Brno, FIT VUT v Brně, 2015

Využití syntézy na systémové úrovni pro aplikace s platformou ZYNQ

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Dr. Ing. Otto Fučíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Husák
25. května 2015

Poděkování

Zde bych chtěl poděkovat vedoucímu práce Doc. Dr. Ing. Otto Fučíkovi za odborné vedení. Také bych rád poděkoval Ing. Petrovi Musilovi, Ing. Martinovi Musilovi a Ing. Filipovi Kadlčkovi za odborné rady a čas při konzultacích.

Obsah

Kapitola 1

Úvod

V dnešní době jsme čím dál více obklopeni elektronikou a počítači. Běžně se setkáváme se stolními počítači, mobilními telefony nebo tablety. Počítače však mohou být vestavěné do různých zařízení, od ledničky až po složité řídící systémy strojů nebo automobilů. Velké nároky na výkon počítače jsou kladeny v oblasti zpracování videa z kamery v reálném čase. K těmto účelům se mohou použít speciální programovatelné počítače. Vývoj programů pro takové počítače není však jednoduchá záležitost. Tato práce zkoumá možnosti využití tzv. syntézy na systémové úrovni, která přináší nový způsob programování těchto počítačů. Tento způsob má za cíl urychlit dobu vývoje a snížit nároky kladené na programátory.

Pro zpracování videa ve vysokém rozlišení lze použít běžný univerzální procesor. Nevýhodou takového přístupu je však velký příkon, rozměry i nedostačující výkon. Rychlost zpracování videa lze urychlit pomocí GPU (Graphic Processing Unit), nicméně příkon a rozměry porostou. Vhodnou technologií pro řešení takových aplikaci je obvod FPGA (Field Programmable Gate Array). FPGA je složeno z matice programovatelných bloků a lze docílit vysoké efektivity při malém příkonu. FPGA se typicky programují pomocí HDL jazyků (Hardware Description Language). Dnes ale existují i nástroje umožňující programovat FPGA pomocí vyšších programovacích jazyků (C/C++). Tyto nástroje umožňují automatické převedení C/C++ algoritmu do HDL popisu obvodu pomocí syntézy na systémové úrovni. Cílem této práce je navrhnout a implementovat aplikaci pro zpracování videa akcelerovanou v FPGA. Návrh FPGA je složen z komponent, které jsou vytvořeny pomocí syntézy na systémové úrovni ve vývojovém prostředí Vivado HLS pomocí jazyka C++. Aplikace je určena pro platformu Xilinx ZYNQ.

Následující kapitola se věnuje teoretickému popisu syntézy na systémové úrovni. Je zde popsán její princip a jednotlivé fáze. Také je zde popsána platforma Xilinx ZYNQ, část ARM procesoru a část programovatelného hradlového pole. Zejména je popsán vývojový kit Xilinx ZC702, který je používán v této práci pro testování aplikace. Poslední oddíl kapitoly se věnuje problematice vývojového prostředí Xilinx Vivado HLS. Třetí kapitola popisuje návrh aplikace, jednotlivé bloky a popis rozhraní mezi nimi. Čtvrtá kapitola se věnuje implementaci a testování. Jsou zde popsány výsledky implementace komponent a způsob testování. Rovněž je zde porovnán přístup vývoje pomocí syntézy na systémové úrovni a přístup pomocí HDL jazyků. Pátá kapitola se věnuje optimalizacím aplikace. Jsou zde popsány optimalizace, které byly použity v rámci vytváření aplikace a jejich vliv na propustnost nebo spotřebu plochy na čipu. Závěrečná kapitola hodnotí výsledky práce a nastiňuje možnosti pokračování.

Kapitola 2

Syntéza na systémové úrovni pro platformu Xilinx ZYNQ

Kapitola obsahuje teoretický úvod k problematice této práce. První část kapitoly se věnuje syntéze na systémové úrovni, jaké jsou její principy a jednotlivé části. Další oddíl se věnuje platformě Xilinx ZYNQ, kde jsou popsány jednotlivé bloky a jejich rozhraní. Poslední část kapitoly popisuje vývojové prostředí Xilinx Vivado HLS. Zde je popsáno, co tento nástroj umožňuje. Také se zde nachází ukázky zdrojových kódů.

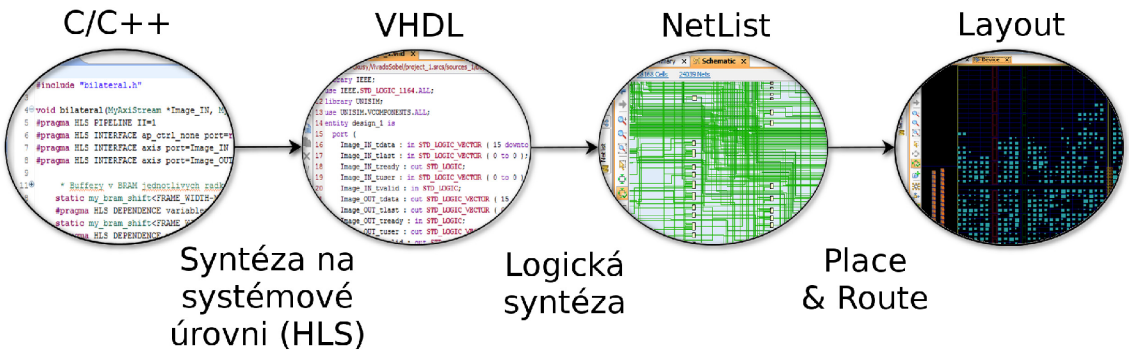
2.1 Syntéza na systémové úrovni

Úkolem syntézy na systémové úrovni (High Level Synthesis - HLS) je najít RTL (Register Transfer Level) realizaci číslicového obvodu, který je specifikován pomocí vysokoúrovňového jazyka. Tento jazyk specifikuje pouze abstraktní chování obvodu. Při syntéze na systémové úrovni rovněž musíme specifikovat určitá omezení a cíle, které musejí být splněny. Obvykle existuje mnoho možných realizací výsledného obvodu, cílem je však najít optimální řešení vzhledem k daným omezením. Například frekvence, plocha na čipu nebo příkon [5].

Obrázek 2.1 znázorňuje vývojový proces od popisu chování číslicového obvodu ve vysokoúrovňovém jazyce až po výslednou konfiguraci (Layout) FPGA čipu. Tato práce se zabývá zejména popisem obvodu pomocí vysokoúrovňového jazyka a rovněž ukáže základní principy syntézy na systémové úrovni. Výstupem syntézy na systémové úrovni je RTL specifikace obvodu (např. v jazyce VHDL nebo Verilog). Pokud na tuto specifikaci aplikujeme logickou syntézu, dostáváme tzv. NetList, což je seznam vzájemně propojených prvků technologie. Tento seznam prvků je nezávislý na cílové technologii. Poté, co definujeme konkrétní čip, je možné rozmístit a propojit (Place & Route) seznam prvků do matice na čipu a zároveň definovat vstupní a výstupní porty. Na konci procesu získáme konfiguraci čipu (Layout).

Motivace k syntéze na systémové úrovni

Motivací pro syntézu na systémové úrovni je zvýšení produktivity vývojářů. Plocha na čipu roste každým rokem a vývoj na úrovni RTL znamená velké nároky na počet vývojářů i na čas. Existují techniky, které zvyšují efektivitu vývoje - například používání existujících bloků (Intellectual Property cores - IP). Další možností je použít syntézu na systémové úrovni. Ta přináší několik výhod pro vývojáře:



Obrázek 2.1: Vývojový diagram při využití HLS.

- **Rychlejší návrh** - Jazyk na vyšší úrovni přináší snadnější návrh a implementaci, než popis chování obvodu na úrovni RTL. Lze to přirovnat k přechodu z assembleru k jazyku C.
- **Efektivnější verifikace a validace obvodu** - Obvod popsaný v jazyce C můžeme snadněji simulovat na běžném procesoru a zároveň popis ve vyšším jazyce je méně náchylný na chyby. Nástroje pro HLS poskytují psaní Test Bench aplikací, které porovnávají výsledky s referenčním algoritmem, což umožní rychlou verifikaci obvodu.
- **Efektivnější ladění obvodu** - Nástroje poskytují možnost ladění na úrovni vyššího jazyka, což umožní rychlejší nalezení chyb v návrhu či implementaci.
- **Snadný průzkum možných mikroarchitektur obvodu** - Nástroje pro HLS umožňují velmi snadno výsledný obvod upravovat podle žádoucích cílů. Můžeme v krátkém čase zjistit například spotřebu zdrojů, latenci nebo propustnost obvodu při různých frekvencích, při rozbalení nebo zřetězení smyček a podobně. Průzkum jednotlivých mikroarchitektur při návrhu na úrovni RTL je o mnoho složitější a časově náročnější.

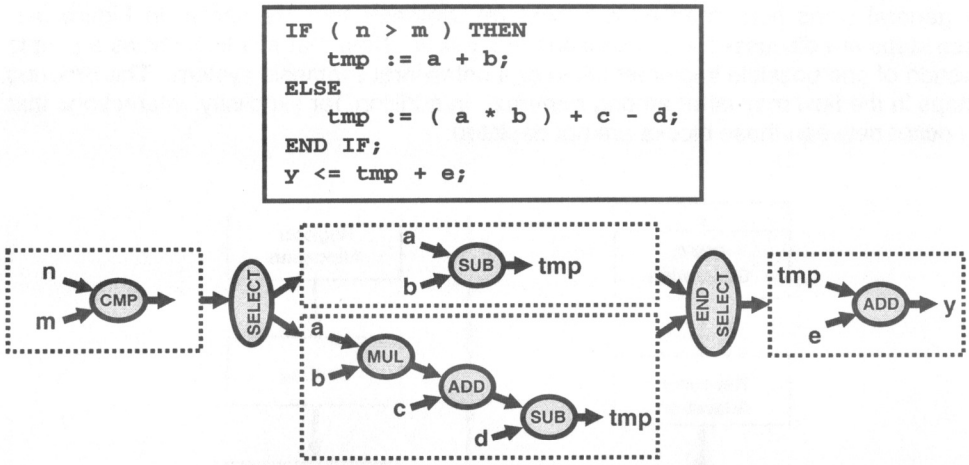
Jak funguje syntéza na systémové úrovni

Syntéza na systémové úrovni má na vstupu popis algoritmu a jejím cílem je vytvořit RTL schéma. Obecně výsledných RTL schémat může existovat více. Syntéza na systémové úrovni má za cíl najít algoritmicky takové schéma, které nejlépe odpovídá vstupním omezením (frekvence, latence, plocha na čipu, ...).

Syntézu na systémové úrovni můžeme rozdělit do několika kroků [2, 6] :

Generování CDFG (Control / Data Flow Graph)

Prvním krokem je převedení vysokoúrovňového jazyka do interní reprezentace - CDFG (Control / Data Flow Graph). Tento graf znázorňuje vstupy a výstupy algoritmu, dále operace, které jsou použity a tok dat od vstupu na výstup. Uzel reprezentuje operaci (MUL, ADD, ...) a hrana datovou závislost mezi operacemi.



Obrázek 2.2: Příklad CDFG grafu vytvořeného z algoritmu. Převzato z [2].

Alokování zdrojů

Při alokování zdrojů se určí počet a typ zdrojů potřebných pro danou implementaci. CDFG znázorňuje pouze datové závislosti, ale neříká nic o zdrojích implementace obvodu. V tomto kroku se zjistí, na kolika bitech se provádí daná operace a předběžně se vyberou možné funkční jednotky. Alokování zdrojů lze také provádět manuálně, kdy programátor určí, jaké zdroje se mají použít.

Plánování

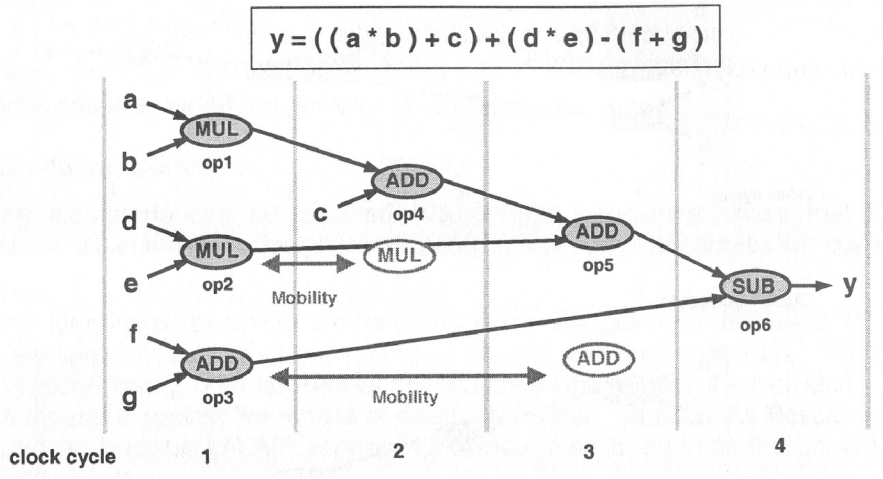
Při této fázi se určí, které operace se vykonají v daném hodinovém taktu. Jedná se o klíčovou fázi, od které se odvíjí kvalita celé systémové syntézy. Komerční nástroje používají uzavřené algoritmy, nicméně zde si nastíníme základní způsob plánování.

Pomocí algoritmů ASAP (As Soon As Possible) a ALAP (As Late As Possible) lze zjistit mobilitu operací. Algoritmus ASAP se snaží naplánovat všechny operace co nejdříve, jak jen to datové závislosti dovolí. Naproti tomu ALAP plánuje operace na co nejpozdější dobu, než budou potřeba k dalšímu výpočtu. Rozdílem těchto dvou hodnot dostáváme mobilitu - to znamená, jaké jsou možnosti posouvání dané operace v rámci hodinových taktů.

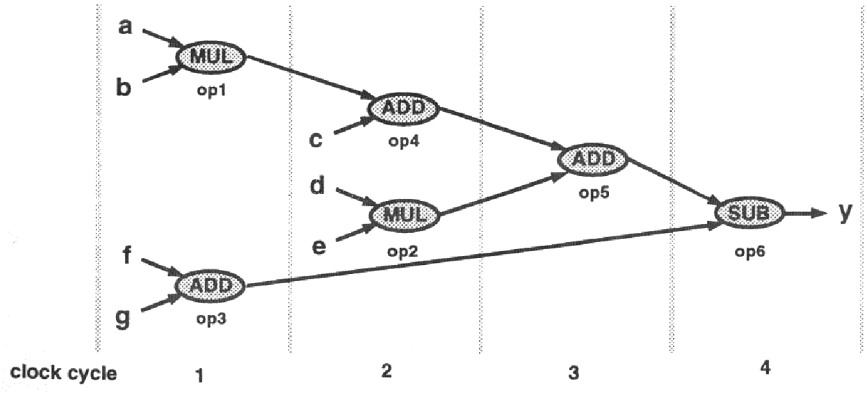
Na obrázku 2.3 vidíme výpočet aritmetického výrazu ve 4 krocích. Například operace `op2` je nyní naznačena v hodinovém taktu 1. Podle mobility se ovšem může vykonat klidně i v taktu 2 a nedojde ke zpomalení výpočtu, protože její výsledek je potřeba až v taktu 3. Rovněž operace `op3` může být posunuta dokonce až do taktu 3 a stále dostáváme celkový výsledek výrazu ve 4 krocích. Cílem plánování je však najít takové rozmístění operací, které bude nejvýhodnější. Prohledávání všech možností však vede k velmi vysokým výpočetním nárokům. Proto se používají algoritmy, které prohledávají možnosti s heuristikou - například Force-Directed Heuristic nebo List-Based Scheduling.

Abychom si ukázali, jaký vliv má plánování na celkový návrh obvodu, ukážeme si na obrázku 2.3 spotřebu zdrojů. Vidíme, že celkové nároky na zdroje budou: 2 MUL, 1 ADD a 1 SUB. Je to proto, že v 1. kroku potřebujeme 2 MUL, které pracují paralelně, pak v dalších krocích je potřeba vždy jen 1 ADD, která se sdílí od 1. do 3.kroku. A ve 4. kroku 1 SUB.

Podíváme-li se na obrázek 2.4, najdeme zde optimální naplánování operací do jednotlivých kroků. Oproti původní verzi je operace `op2` přesunuta do 2. kroku. Výsledné požadavky



Obrázek 2.3: Mobilita operací po vypočítání algoritmy ASAP a ALAP. Převzato z [2].



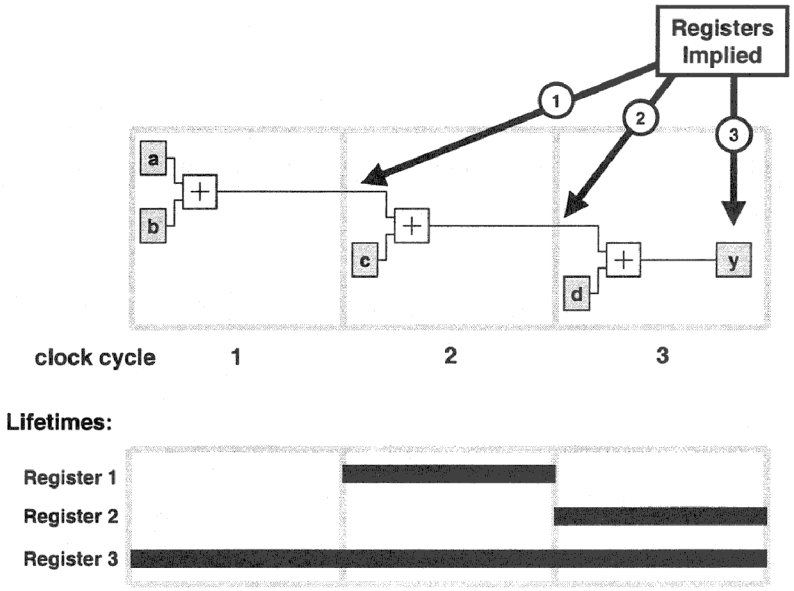
Obrázek 2.4: Optimálně naplánované operace. Převzato z [2].

na zdroje budou následující: 1 MUL, 1 ADD a 1 SUB. Nyní použijeme jen jednu násobičku, která bude sdílená pro operace `op1` a `op2`. Ve výsledku jsme tedy ušetřili jednu násobičku. Lze vidět, že na výslednou realizaci obvodu má velký vliv plánování operací.

Alokace registrů

V této fázi se řeší alokace registrů, do kterých se ukládají výsledky operací. Výsledná data musí být uložena v registrech, ale každý výsledek má různou dobu života. Proto je důležité přiřadit, kam se jednotlivé výsledky budou ukládat. Také je potřeba zjistit počet celkově potřebných registrů. Pro zjištění počtu registrů pro alokaci se používá Left-Edge algoritmus.

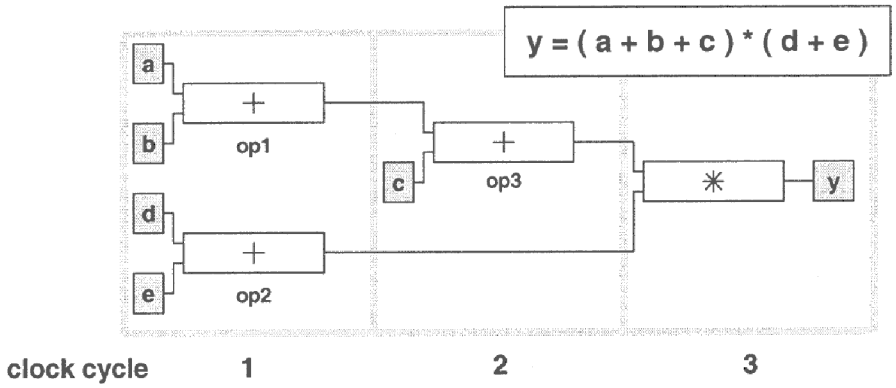
Obrázek 2.5 zobrazuje obvod, který má na výstupu proměnnou `y`. Když je spočítaná nová hodnota, tak je výstupní proměnná aktualizována. Proto výstupní proměnná má dobu života přes všechny hodinové takty. Navíc bude potřeba už jen jeden registr, protože registr 1 a 2 je sdílený.



Obrázek 2.5: Doba života registrů. Převzato z [2].

Přiřazení

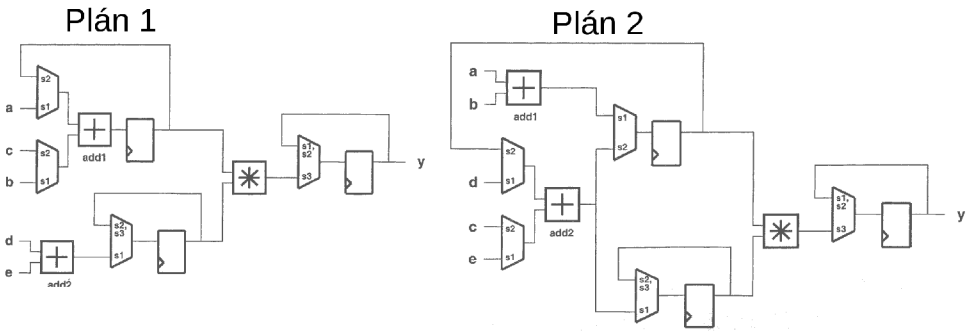
Přiřazení je proces, který každé naplánované operaci přiřazuje funkční jednotku. Cílem je opět dosáhnout co nejjednoduššího výsledného obvodu. Následující příklad přiřazení ukazuje, že záleží i na zvolené funkční jednotce, která provádí operaci.



Obrázek 2.6: Jednoduchý naplánovaný obvod. Převzato z [2].

Na obrázku 2.6 lze vidět opět jednoduchý výpočet, který již je naplánovaný - jsou přiřazeny operace do jednotlivých hodinových taktů. Ovšem ještě nevíme, které funkční jednotky budou vykonávat dané operace. Z obrázku je patrné, že budou potřeba 2 sčítačky, protože v prvním kroku se paralelně provádí operace `op1` a `op2`. Ovšem operace `op3` je již sama v taktu 2 a může být tedy vykonána jednou z dostupných sčítaček. Intuitivně se může zdát, že na tom nezáleží, nicméně pokud zkusíme obě možnosti přiřazení, dostáváme rozdílné výsledky.

Na obrázku 2.7 lze vidět, že první varianta ušetřila jeden multiplexor a spoj. K algoritmickému zjištění, jaká varianta je výhodnější, se používá algoritmus založený na dekompo-



Obrázek 2.7: Dva možné plány přiřazení operací k funkčním jednotkám. [2]

zici grafu na kliky.

Generování RTL

Podle výsledků předchozích kroků jsou vygenerovány datové cesty a řadič - konečný automat (FSM), který řídí obvod. FSM aktivuje funkční jednotky, nastavuje signály a řídí multiplexory. Na konci procesu syntézy na systémové úrovni je vygenerováno výsledné RTL zapojení, například ve VHDL nebo Verilog kódu.

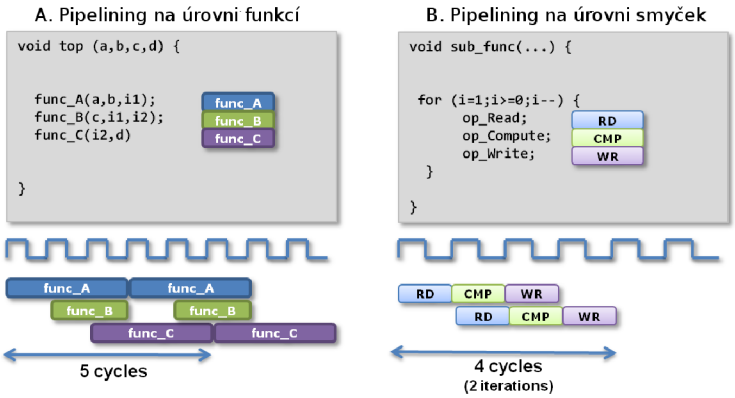
Hlavní optimalizace návrhu

Pokud při návrhu obvodu použijeme zdrojový kód určený pro běžný procesor, dostaneme po syntéze na systémové úrovni obvod, který zpravidla nesplňuje požadavky, které očekáváme. Obvod sice bude logicky ekvivalentní a bude dávat validní výsledky, ale pravděpodobně již nebudou splněny požadavky na propustnost, latenci a zejména na plochu na čipu. Nyní si ukážeme základní principy, které se používají k optimalizaci kódu při návrhu obvodu [1, 9].

- **Optimalizace propustnosti** - zřetězení (pipelining), rozbalení smyček (unrolling), rozdělení polí (partitioning arrays)
- **Optimalizace latence** - slučování smyček (merging), flattening u vnořených smyček
- **Optimalizace plochy na čipu** - použití bitových datových typů, vložení funkcí (inlining), mapování menších polí do velkého pole

Zřetězení (Pipelining)

Zřetězení je často používanou technikou při návrhu digitálních obvodů. Cílem zřetězení je dosáhnout vyšší propustnosti obvodu díky paralelnímu zpracovávání. Při zřetězení se daná kombinační síť rozdělí na menší bloky a s ohledem na datové závislosti se naplánují jednotlivé bloky, které lze vykonat současně. Při syntéze na systémové úrovni lze zřetězení aplikovat na funkce nebo smyčky. Parametrem zřetězení je tzv. inicializační interval (označovaný II), který definuje propustnost zřetězené linky. Tedy kolik taktů hodinového signálu trvá, než je načten nový vstup do linky. Například při inicializačním intervalu 1 (II=1) načítáme každý takt novou hodnotu do linky. Také každým taktem dostáváme na výstupu spočtenou hodnotu. Dalším parametrem je latence, která udává délku linky v počtu hodinových taktů.



Obrázek 2.8: Zřetězení na úrovni funkcí a smyček. Převzato z [9].

Rozbalení smyček (Unrolling)

Rozbalování smyček je technika, kdy rozepíšeme tělo cyklu N-krát za sebe. Několik cyklů se provede paralelně, namísto iterativního zpracovávání ve smyčce. Výhodou této techniky je zvýšení propustnosti, nevýhodou však je výrazné zvýšení požadavků na plochu čipu. Úplné rozbalení smyček lze provádět pouze u cyklů, kde dopředu víme počet iterací. Kompromisem mezi požadavky na zdroje a propustností může být částečné rozbalení smyčky. V tomto případě se udává počet iterací, které se rozbalí, ale zároveň bude zachován cyklus.

Rozdělení polí (Partitioning Arrays)

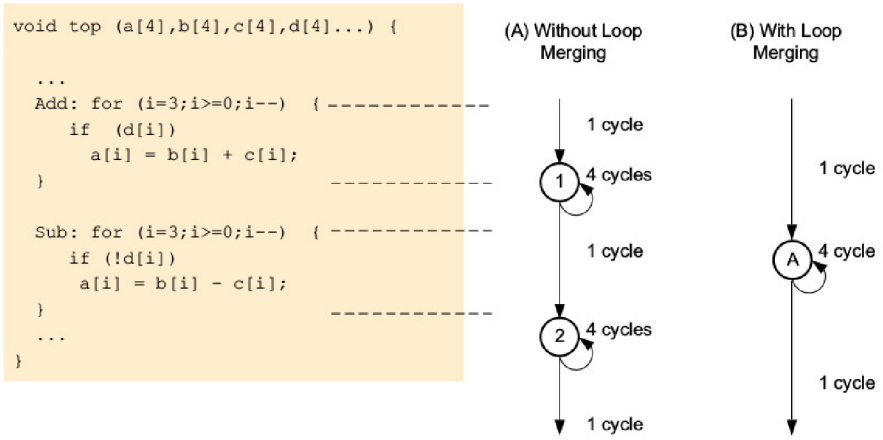
Technika rozdělování se používá v případě, kdy máme pole a pouze omezený počet portů pro čtení. Typicky se jedná o pole uložené v BRAM (Block RAM). Původní pole lze rozdělit na menší pole a tím dosáhnout většího počtu paralelních přístupů k hodnotám. Rozdělení polí má několik možností. Originální pole lze rozdělit do menších po blocích nebo cyklicky. Další možností je kompletní rozdělení, kdy každý prvek pole je přesunut do registrů.

Slučování smyček (Merging)

Slučování smyček je technika, kdy se snažíme sloučit těla podobných cyklů do jednoho cyklu. Při použití slučování na běžném CPU nedosáhneme velkého přínosu. Ovšem při syntéze na systémové úrovni na FPGA má každá smyčka svůj řídící konečný automat (Finite State Machine - FSM), který bude vytvořen při sloučení pouze jednou. Tím dojde k ušetření plochy na čipu.

Flattening u vnořených smyček

Pokud máme vnořenou smyčku uvnitř jiné smyčky, typicky u průchodu dvourozměrnou maticí, přidávají se takty režie při inicializaci a ukončení vnitřní smyčky. Technika flattening odstraňuje tyto takty navíc a sloučí obě smyčky do jedné, podobně jako technika slučování smyček (merging).



Obrázek 2.9: Slučování smyček. Převzato z [9].

Bitové datové typy

Při programování na CPU si můžeme zvolit z několika datových typů. Jedná se zejména o celočíselné hodnoty na 8, 16, 32 a 64 bitů nebo čísla s pohyblivou řádovou čárkou na 32 a 64 bitů. Při programování FPGA však lze zvolit libovolnou datovou šířku proměnných. Pokud v algoritmu určíme, kolik bitů dostačuje na danou proměnnou, ušetříme tím nezanedbatelné místo na čipu.

Pokud potřebujeme použít desetinné číslo, v mnoha případech stačí použít čísla s pevnou řádovou čárkou. Jedná se o číslo, které má pevný počet bitů pro celou a pak pro desetinnou část. Pracování s pevnou řádovou čárkou spotřebuje méně zdrojů, než při použití float proměnných v obvodu. Často lze také použít operace nad celými čísly a následně využít bitové posuny, které umožní ještě nižší spotřebu zdrojů.

Vkládání funkcí (Inlining)

Technika odstraňuje hierarchii volání funkcí. Tělo volané inline funkce je vloženo do místa volání. Vložené tělo funkce pak může být sloučeno a optimalizováno s volající funkcí.

Dostupné nástroje pro syntézu na systémové úrovni

Práce se zabývá nástrojem Xilinx Vivado HLS. Na trhu ovšem existují i další, především komerční nástroje. V této podkapitole si představíme některé příklady:

Vivado Design Suite - Nástroj od firmy Xilinx[1], obsahuje integrované prostředí pro návrh, ladění a simulaci vlastních IP-core.

Catapult C - Nástroj od firmy Calipto[2], obsahuje rovněž nástroje pro modelování, syntézu a verifikaci pro čipy ASIC a FPGA od různých firem. Umožňuje psát program v jazyce ANSI C++ a SystemC.

CyberWorkBench - Nástroj od firmy NEC[3], jedná se o komerční nástroj, který rovněž poskytuje nástroje pro syntézu, simulaci a verifikaci. Je založen na jazyce C. RTL výstup je pro čipy FPGA a ASIC od různých firem.

[1] http://www.xilinx.com/products/design-tools/vivado.html
[2] http://calypto.com/en/products/catapult/overview/
[3] http://www.nec.com/en/global/prod/cwb/

Leg Up - Nástroj z University of Toronto[4], jedná se o open source nástroj a zaměřuje se na syntézu z jazyka C na Verilog.

Programovací jazyky pro syntézu na systémové úrovni

Programování pro syntézu na systémové úrovni se provádí v jazycích založených na jazyku C. Xilinx Vivado podporuje jazyky C/C++ i jazyk SystemC [1].

Jazyk C - univerzální a dnes velmi rozšířený programovací jazyk. Vivado HLS obsahuje knihovny rozšiřující tento jazyk o možnost práce s bitovými typy a plovoucí a pevnou řádovou čárkou.

Jazyk C++ - Jazyk C++ je objektově orientovaný jazyk a rozšiřuje jazyk C o další konstrukce - třídy, šablony, polymorfismus nebo virtuální třídy. Obzvláště šablony se často používají při vývoji. Umožňují definovat parametrizovatelné datové struktury, což zpřehledňuje zdrojový kód.

Jazyk SystemC - Tento jazyk rozšiřuje C++ o knihovnu `systemc.h`, která umožňuje popsat obvod i s principy HDL jazyků - popis hierarchie, paralelismu nebo vyjádření času. Jazyk SystemC je používán rovněž pro modelování a verifikaci interakcí jednotlivých komponent systému na čipu (Transaction Level Modelling (TLM), Electronic System-Level (ESL)).

Další výrobci softwaru pro syntézu na systémové úrovni používají i jiné jazyky. Je to například HandleC, ESL nebo MATLAB.

Srovnání syntézy na systémové úrovni s HDL jazyky

Existují také vědecké články, které se zabývají srovnáním syntézy na systémové úrovni s implementací pomocí HDL jazyků. Jeden z článků [7] se věnuje porovnání implementace Lloydova algoritmu, který hledá nejbližší střed v prostoru a používá se v K-means algoritmu. Výsledky jsou v následující tabulce 2.1. Výsledky jsou z testu nad 16384 vzorky, které se přiřazovaly ke 128 středům.

	REG	LUT	BRAM	DSP	perioda [ns]	cykly/iteraci	čas/iteraci [us]
VHDL	62168	66472	83	120	5.0	53k	264
HLS	63878	68360	89	120	9.7	66k	637

Tabulka 2.1: Porovnání Lloydova algoritmu - VHDL implementace a HLS implementace.

Další diplomová práce [13] se věnuje srovnáním syntézy na systémové úrovni a HDL popisu obvodu. V této práci byl implementován rozmazávající filtr s oknem 9x9 jak pomocí HLS, tak pomocí VHDL. V následující tabulce 2.2 jsou shrnuty výsledky.

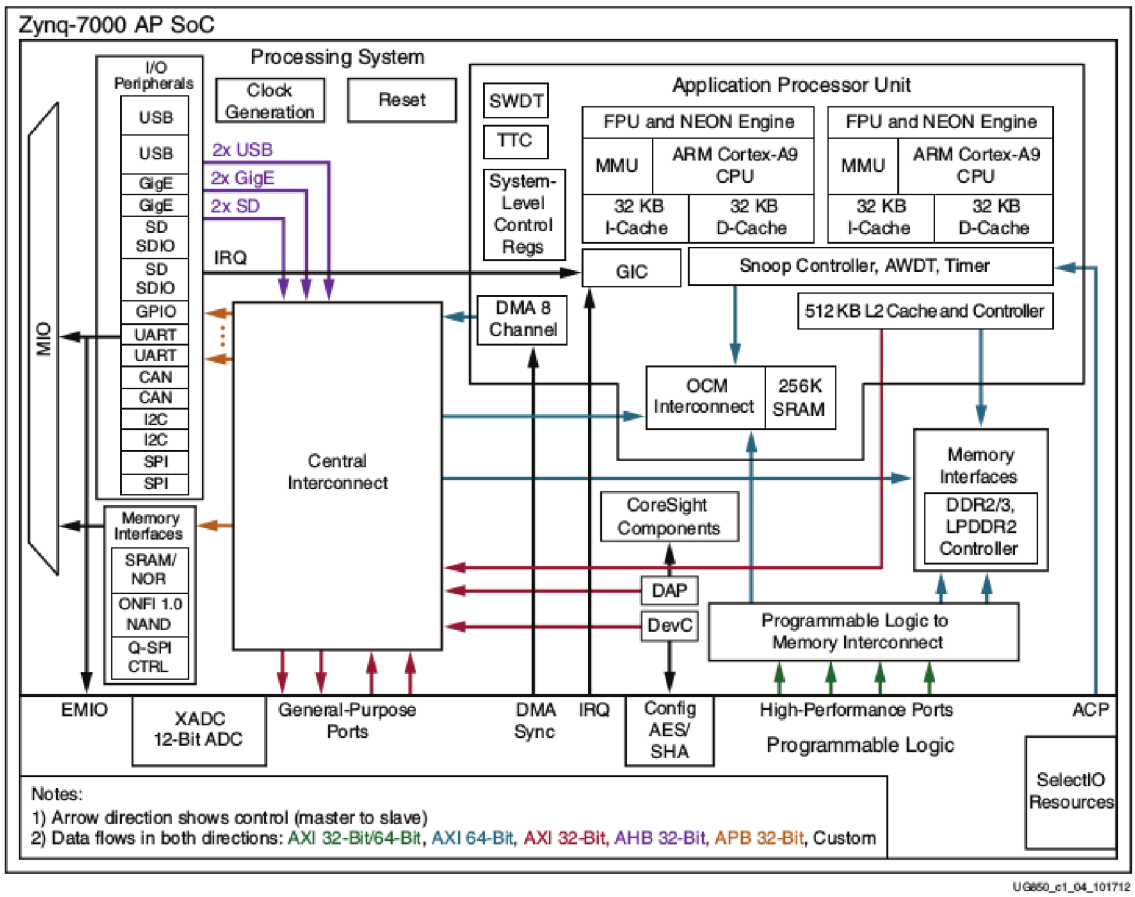
	FF	LUT	BRAM	DSP	čas vývoje
VHDL	6139	2989	12	0	33 h
HLS	5970	4827	12	0	15 h

Tabulka 2.2: Porovnání rozmazávajícího filtru s maskou 9x9 - VHDL implementace a HLS implementace.

[4] http://legup.eecg.utoronto.ca/

2.2 Platforma Xilinx ZYNQ

Systém na čipu (System On Chip - SoC) Xilinx Zynq kombinuje FPGA a aplikační procesor v jednom obvodu. Jméno Zynq údajně dostal podle chemického prvku zinek, který je možné směšovat s různými dalšími kovy. Rovněž Xilinx Zynq má díky své flexibilitě celou škálu uplatnění. Obsahuje dvoujádrový ARM Cortex-A9 procesor a programovatelné hradlové pole (FPGA). Na procesoru může být spuštěn operační systém Linux. Architektura celého systému na čipu je propojena sběrnicí AXI (Advanced eXtensible Interface) [1]. Na obrázku 2.10 lze vidět jednotlivé bloky systému na čipu. Zynq se skládá z dvou hlavních částí: Processing System (PS) a Programmable Logic (PL). V následujících podkapitolách si blíže popíšeme tyto části, rovněž sběrnici uvnitř čipu a vstupní a výstupní rozhraní. Poté se seznámíme s vývojovou deskou Xilinx ZC702 a s možným využitím zařízení.



Obrázek 2.10: Blokový diagram SoC Xilinx Zynq. Převzato z [11].

Processing system - PS

Zynq obsahuje dvoujádrový procesor ARM Cortex-A9. Jedná se o "hard"procesor, který je přímo na čipu. Alternativou mohou být "soft"procesory (např. MicroBlaze), které jsou v FPGA jako IP blok. Použití soft procesorů je flexibilní a lze jich v FPGA umístit několik. Výhodou hard procesorů však je, že dosahují vyššího výkonu. Processing system neobsahuje jen procesor ARM, ale, jak lze vidět na obrázku 2.10, i další jednotky - Application

Processing Unit, paměť cache, rozhraní k paměti, generátor hodin, a řadu rozhraní.

Application Processing Unit (APU) obsahuje dvě jádra ARM procesoru s přídavnými jednotkami pro akceleraci - NEON Media Processing Engine pro akceleraci práce s multimédii (kódování videa apod.), FPU pro práci s plovoucí řádovou čárkou, Memory Management Unit, L1 a L2 cache. ARM dokáže běžet až na 1GHz (záleží však na konkrétním Zynq zařízení). Každé jádro má 32kB L1 cache, 512kB sdílené L2 cache a ještě 256kB OCM (On Chip Memory).

K Processing system mohou být připojeny rovněž externí periferie přes rozhraní MIO (Multiplexed Input/Output). Jedná se o 54 pinů, které mohou být mapovány přímo na fyzické piny procesoru Zynq. Pokud však je potřeba připojit více periferií, musí se připojit přes EMIO (Extended MIO). EMIO není mapováno přímo na fyzické piny, ale je připojeno k PL (Programmable Logic) a poté je lze namapovat na fyzické piny. Processing system obsahuje tyto periferie:

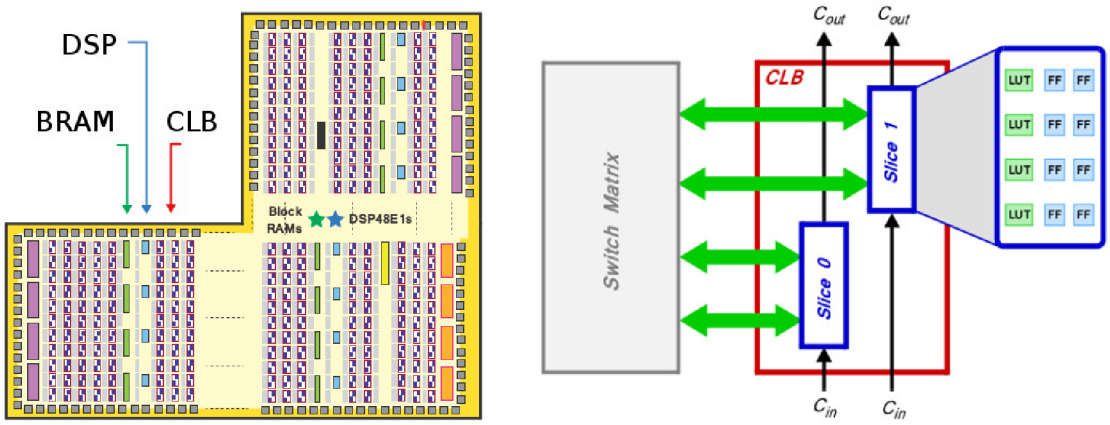
- **SPI** - Serial Peripheral Interface. Standardní sériová komunikace, založené na 4 pinovém rozhraní.
- **I2C** - Inter-Integrated Circuit. Nízkorychlostní sběrnice, 2 pinové rozhraní. Podpora master a slave módu.
- **CAN** - Controller Area Network. Sériová sběrnice dosahující rychlosti až 1Mb/s, používaná ke komunikaci mezi prvky systému, uplatněná zejména v automobilovém průmyslu.
- **UART** - Universal Asynchronous Receiver Transmitter. Nízkorychlostní sběrnice, často používaná pro připojení terminálu k PC.
- **GPIO** - General Purpose Input/Output. Univerzální vstupy a výstupy. Používá se pro LED, tlačítka nebo přepínače. Obsahuje 4 banky po 32 bitech.
- **SD** - Rozhraní pro Secure Digital kartu.
- **USB** - Universal Serial Bus. Podpora USB 2.0, lze používat jako host, zařízení nebo kombinovaně (On The Go -OTG mód).
- **Ethernet** - Ethernetové rozhraní, podpora pro 10 Mbps, 100 Mbps a 1 Gbps.

Programmable Logic - PL

Druhá část architektury procesoru Zynq je Programmable Logic. Je založena na FPGA Xilinx 7. řady - Artix nebo Kintex (záleží na variantě Zynq procesoru). Programmable Logic je složeno z matice CLB (Configurable Logic Blocks), ze vstupního a výstupního rozhraní, z přepínací a přenosové logiky a také ze speciálních bloků - DSP (Digital Signal Processor) a BRAM komponent.

Nyní budou popsány jednotlivé zdroje a také kolik se jich nachází v jednotlivých variantách čipu Zynq.

- **LUT (Lookup Table)** - tento flexibilní prvek může být použit čtyřmi způsoby: 1. logická funkce s až 6 vstupy, 2. malá Read Only Memory (ROM), 3. malá Random Access Memory (RAM), 4. posuvný register (Shift Register LUT - SRL)



Obrázek 2.11: Struktura PL (vlevo) a detail CLB (vpravo). Převzato z [1].

- **FF (Flip-flop)** - bistabilní klopný obvod, používá se k implementaci jednobitového registru.
- **BRAM (Block RAM)** - tyto paměti rozmístěné na čipu mezi CLB obsahují každá 36Kb. Paměť lze také použít jako 2 nezávislé bloky po 18Kb. BRAM je určená pro ukládání větších dat na čipu. Alternativou je použití LUT paměti, která ale spotřebuje více místa na čipu. BRAM je vhodná pro implementaci RAM, ROM a FIFO pamětí. Každá BRAM obsahuje 2 porty, pro čtení nebo zápis. Například při implementaci FIFO paměti je jeden port pro čtení a druhý pro zápis. Pokud je potřeba číst nebo zapisovat více prvků naráz, je potřeba využít paměti LUT.
- **DSP48** - používá se pro implementaci rychlých aritmetických operací, které mají střední až dlouhou velikost slova. Pro krátké bitové operaci se používají logické funkce v LUT. DSP obsahuje pre-adder/subtractor, násobičku a post-adder/subtractor. Post-adder/ substracotr obsahuje rovněž logické funkce (NOT, AND, OR, NAND, NOR, XOR, a XNOR). Výsledek je na 48 bitech. DSP může být taktované na maximální frekvenci zařízení při malých nárocích na příkon.

	Z-7010	Z-7015	Z-7020	Z-7030	Z-7045	Z-7100
PL	Artix-7			Kintex-7		
FF	35 200	96 400	106 400	157 200	437 200	554 800
LUT	17 600	46 200	53 200	78 600	218 600	277 400
BRAM 36Kb	60	95	140	265	545	755
DSP48	80	160	220	400	900	2020

Tabulka 2.3: Přehled variant Zynq procesorů s počty zdrojů.

Rozhraní mezi PS a PL

Na obrázku 2.10 je vidět, že mezi Processing System a Programmable Logic existují rozhraní. Tyto komunikace jsou založeny na standardu AXI. Standard AXI je část standardu ARM AMBA, který byl vyvinut pro komunikaci v mikrokontrolérech a dnes má uplatnění

také v systémech na čipu. AXI standard obsahuje více protokolů, jejich výběr záleží na požadované aplikaci.

- **AXI4** - používá se pro linky mapované do paměti, pro vysokorychlostní přenosy. Data posílá v dávce až 256 slov.
- **AXI4-Lite** - zjednodušená linka, používá adresaci do paměti. Data jsou přenesena od zdroje do paměti. Neposílá se v dávce, ale zvlášť.
- **AXI4-Stream** - používá se pro vysokorychlostní streamování dat. Neprobíhá zde adresace, ale data tečou přímo od zdroje k cíli.

Rozhraní mezi PS a PL má více kanálů. PS obsahuje Central Interconnect a Memory Interconnect, což jsou přepínače, které spravuje více kanálů rozhraní. K Central Interconnect jsou připojeny General Purpose porty. Je to 32b sběrnice, která je vhodná pro nízké a středně rychlé přenosy. Tato rozhraní neobsahují vyrovnávací paměti. Rozhraní jsou celkem čtyři (2x je master PS, 2x je master PL). K Memory Interconnect jsou připojeny čtyři High Performance porty. Každý obsahuje vyrovnávací paměť FIFO. Pracuje v dávkovém režimu a je vhodné pro vysokorychlostní přenosy mezi PL a pamětí v PS. Datová šířka je 32 nebo 64 bitů, PL je vždy master. Dalším rozhraním mezi PS a PL je EMIO, které rozšiřuje možnost připojení externích zařízení.

Vývojová deska Xilinx ZC702

Firma Xilinx nabízí vývojářům několik vývojových desek. Jednou z nich je Xilinx ZC702 [10]. Deska obsahuje mnoho rozhraní a je osazena čipem Xilinx Zynq 7020. K procesoru Zynq je připojena 1GB DDR3 paměť. Programování lze provádět přes JTAG. Na vývojové desce se nachází tyto periferie: 1 Gb ethernet, USB 2.0, UART, GPIO - programovatelné vstupy a výstupy, I2C, CAN Bus, HDMI pro výstup na monitor, SD karta. Také se zde nachází LED a uživatelská tlačítka a přepínače [10].

Na desce se nachází 16MB Quad SPI Flash, která lze využít i pro uložení programu a lze využít jako bootovací médium. Pokud však je používán OS Linux nebo velikost této paměti nestačí, používá se pro uložení programu SD karta a lze z ní bootovat. Na desce je také oscilátor pro generování systémových hodin o maximální frekvenci 200 MHz.

Kartu lze rozšířit o další periferie přeš FMC sloty (FPGA Mezzanine Connectors). K těmto konektorům lze připojit zařízení jako vícekanálové ADC karty, připojení SATA, 3G modulů nebo rozšíření o programovatelné vstupy a výstupy.

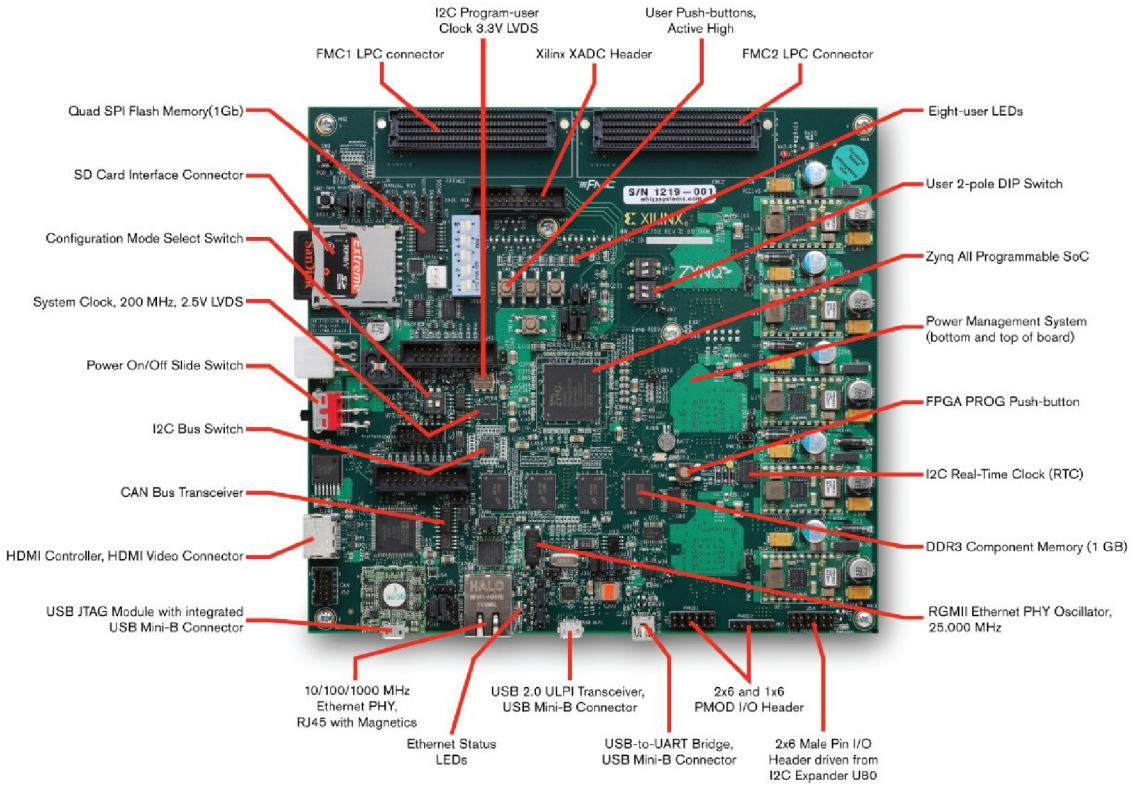
Kromě vývojové desky Xilinx ZC702 existují i další desky s čipem Zynq. Je to například deska Xilinx ZC706[5], Zedboard[6] nebo ZYBO[7].

Příklady využití SoC Xilinx ZYNQ

V této podkapitole si ukážeme praktické příklady využití systému na čipu Xilinx ZYNQ:

- **Zpracování obrazu a videa** - Jedná se o aplikace zahrnující kamery, zařízení na komprimaci videa a úložný prostor. Zynq zde může být použit pro filtrování obrazu a různé úlohy z počítačového vidění v reálném čase.

[5] http://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html
[6] http://zedboard.org/
[7] http://www.xilinx.com/support/university/boards-portfolio/xup-boards/XUPZYBO.html

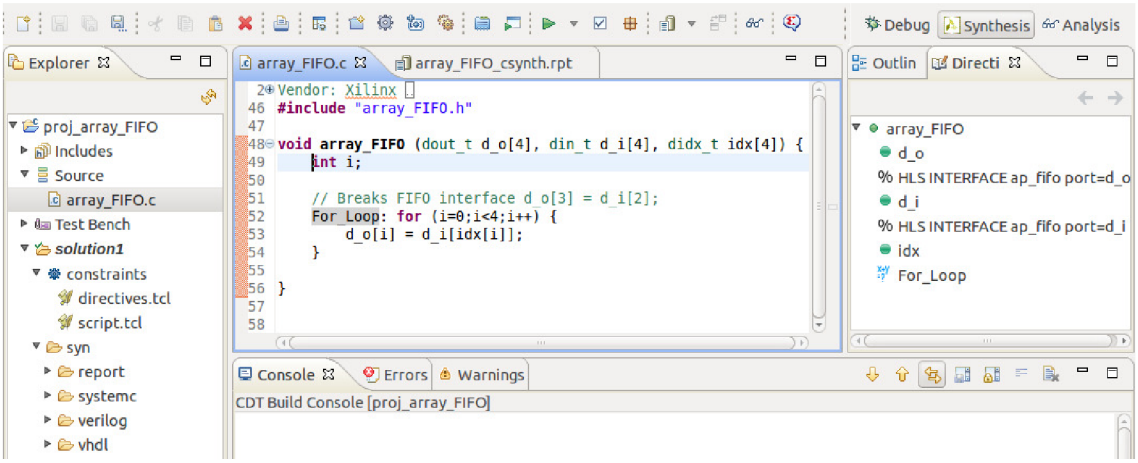


Obrázek 2.12: Vývojová deska Xilinx ZC702 [10].

- **Komunikace** - V této oblasti se jedná o bezdrátovou i drátovou komunikaci - například satelitní přenosy, radary, sonary, GPS a také aplikace pracující se síťovými pakety.
- **Řídící systémy, robotika** - Oblast zahrnuje automobilový průmysl, kde se používají senzory pro zvýšení bezpečnosti a pohodlí v automobilech. Rovněž lze Zynq použít pro obranu a letectví, kde může řídit navigační a komunikační systémy v reálném čase. Dále například řízení výrobních procesů v průmyslu nebo v medicíně zpracovávání signálů z ultrazvuků nebo magnetické rezonance.

2.3 Vývojové prostředí Xilinx Vivado Design Suite

V této kapitole budou předvedeny praktické postupy, jak vyvíjet aplikace pro platformu Xilinx Zynq s využitím syntézy na systémové úrovni. Jako vývojové prostředí budeme používat Xilinx Viado Design Suite, jehož součástí je jednak aplikace Vivado, Xilinx SDK a potom také Vivado HLS. Aplikace Vivado je určena pro vytváření návrhu pro FPGA, provádí se zde návrh obvodu, logická syntéza, implementace a generování bitstreamu. Prostředí Xilinx SDK slouží k programování ARM procesoru. Konečně aplikace Vivado HLS, které se budeme nejvíce věnovat slouží k vytváření komponent ve vysokoúrovňovém jazyce a za použití syntézy na systémové úrovni je vygenerována komponenta s RTL popisem obvodu. Následně tato komponenta se dá umístit do návrhu v aplikaci Vivado a provést logickou syntézu.



Obrázek 2.13: Vývojové prostředí Vivado HLS

Na obrázku 2.13 můžeme vidět prostředí aplikace Vivado HLS. Aplikace je logicky rozdělena do 3 režimů, které lze přepínat v pravém horním rohu aplikace.

- **Debug** - slouží ke klasickému ladění kódu. V tomto režimu můžeme sledovat proměnné, přidávat breakpointy a ladit algoritmus napsaný ve vysokoúrovňovém jazyce.
- **Synthesis** - režim je určený pro optimalizování algoritmu pro vysokoúrovňovou syntézu. Do kódu můžeme vkládat direktivy, můžeme spouštět simulaci pomocí Test Bench souborů a hlavně můžeme spouštět syntézu a generovat RTL popis obvodu.
- **Analysis** - zde lze analyzovat výsledné řešení obvodu. Najdeme zde v tabulkách rozepsané potřebné zdroje pro řešení. Také je zde tabulka, kde jsou rozepsané operace do jednotlivých hodinových taktů.

Při vývoji aplikace a zjišťování, která mikroarchitektura je výhodnější, je dobré používat více řešení (Solutions). Při vývoji si můžeme vytvořit několik řešení s jinými parametry a poté porovnat metriky, které nás zajímají. Direktivy pro syntézu na systémové úrovni můžeme psát buď přímo do zdrojového kódu pomocí #pragma HLS (v této práci to bude pro přehlednost použito) a nebo lze je psát do TCL skriptu. To je vhodné použít při použití více řešení, kde každé řešení má svůj TCL skript a přitom soubor se zdrojovým kódem zůstane stejný pro všechna řešení. Direktivy lze přidávat v pravé části okna v režimu syntézy.

Postup vytvoření komponenty

Nyní budou ukázány jednotlivé kroky, které je třeba pro vytvoření komponenty v aplikaci Vivado HLS a její integrace do celkového návrhu v prostředí Vivado.

Vytvoření projektu a definice chování komponenty

Během vytváření nového projektu je potřeba zadat požadovanou periodu hodin (lze zadat i jako frekvenci např. 150 MHz) a definovat cílový čip nebo vývojovou desku. Zdrojový kód se může obecně skládat z více funkcí, ale jedna musí být označena jako top funkce. Tato funkce poté definuje rozhraní komponenty. Jaké jsou možné typy rozhraní je napsáno v kapitole 2.3. Při psaní programu lze také využít knihovnu HLS C Library 2.3, která obsahuje datové typy a třídy upravené pro syntézu. Pro optimalizaci kódu a specifikaci chování se využívají direktivy pro syntézu na systémové úrovni. Kompletní seznam všech direktiv je v kapitole 2.3.

C simulace pomocí Test Bench

Kromě zdrojových souborů je možné také do projektu přidat Test Bench soubor pro testování algoritmu. Tento soubor musí obsahovat funkci main(). Často se používá technika simulací, kdy máme tzv. golden algoritmus v jazyce C a pak algoritmus upravený pro syntézu na systémové úrovni. Poté spustíme vstupní hodnoty do obou algoritmů a porovnáváme výsledky, zda upravený algoritmus odpovídá referenčnímu.

C simulaci spouštíme tlačítkem `Run C Simulation`. Poté si můžeme vybrat, zda chceme spustit i Debugger. Tímto způsobem lze efektivně testovat algoritmus bez nutnosti provádět syntézu a případně nahrávat řešení do vývojové desky.

Analýza syntézy

Po stisku tlačítka `C Synthesis` se spustí syntéza na systémové úrovni pro daný projekt. Výpis během syntézy lze vidět v okně Console ve spodní části aplikace. Syntéza u menších projektů trvá desítky vteřin u větších několik minut. Po syntéze se objeví okno s výpisem, kde jsou shrnuty výsledky požadovaných zdrojů, jaké periody hodin lze dosáhnout a jaká je latence a interval komponenty. Pokud se některá hodnota nesplnila, objeví se v tomto výpise červeně. Ukázku výpisu lze vidět na obrázku 2.14.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	444
FIFO	-	-	-	-
Instance	-	4	145	913
Memory	4	-	0	0
Multiplexer	-	-	-	81
Register	-	-	480	1
Total	4	4	625	1439
Available	280	220	106400	53200
Utilization (%)	1	1	~0	2

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	6.67	5.73	0.83

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
16	16	1	1	function

Obrázek 2.14: Výsledky syntézy na systémové úrovni.

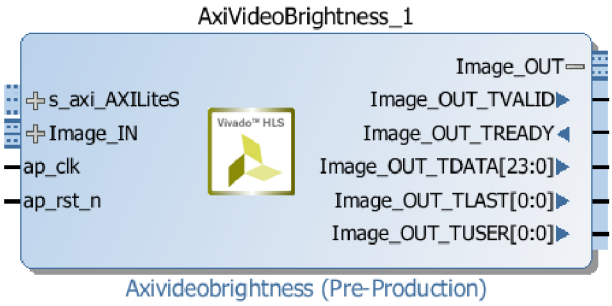
Po úspěšně provedené syntéze lze přejít do režimu analýzy. Zde je podrobně vidět naplánované operace v jednotlivých taktech hodinového signálu. Na obrázku 2.15 lze vidět rozepsané operace během 17-ti taktů zpracování jednoho pixelu. Po kliknutí na danou operaci můžeme zobrazit zdrojový kód ve vysokoúrovňovém jazyce.

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1	⊟I/O Ports																	
2	Image_IN_last_V	read																
3	Image_IN_user_V	read																
4	Image_IN_data_V	read																
5	Image_OUT_data_V																	write
6	Image_OUT_user_V																	write
7	Image_OUT_last_V																	write
8	⊟Instances																	
9	sobel_mul_19s_19s_25_5_U2				*													
10	sobel_mul_19s_19s_25_5_U1				*													
11	grp_sobel_fxp_sqrt_18_18_25_20_s_...									call								
12	⊟Memory Ports																	
13	buffer1_data_V(p0)	read																
14	buffer0_data_V(p0)		read															
15	buffer1_data_V(p1)			write														
16	buffer0_data_V(p1)			write														
17	⊟Expressions																	
18	tmp_11_fu_256	+																
19	storemerge5_fu_262	se...																

Obrázek 2.15: Analýza syntézy algoritmu.

Export a integrace komponenty

Už jsme dosáhli požadovaných vlastností komponenty a nyní je potřeba vygenerovat RTL pomocí tlačítka `Export RTL`. Tím se ve složce projektu a dané implementace vytvoří složka ip obsahující .zip soubor, kde je komprimovaná celá komponenta.



Obrázek 2.16: Komponenta v aplikaci Vivado.

Nyní je třeba spustit aplikaci Vivado, kam můžeme do návrhu umístit komponentu. Nejprve musíme přidat komponentu do seznamu IP Catalog. Spustíme IP Catalog → IP Setting → IP (záložka) → Add Repository. Zde je možné přidat adresář s komponentou. Pokud ve Vivado HLS exportujeme novou verzi komponenty, stačí v záložce IP stisknout tlačítko Refresh Repository. Poté je třeba spustit Repository IP Status (Tools → Report → Repository IP Status) a následně stisknout Upgrade u označených komponent, které chceme aktualizovat v návrhu.

Knihovna High-Level Synthesis C

Vivado HLS rozšiřuje jazyk C o knihovnu HLS C Libraries. Obsahuje funkce a datové typy, které jsou jednoduše syntetizovatelné. V následujících příkladech budeme v této práci používat jazyk C++ [9].

Arbitrary Precision Data Types Library

V kapitole 2.1 byly představeny bitové datové typy. U celočíselných typů se parametrizuje datová šířka a znaménko. U pevné řádové čárky se definuje datová šířka proměnné, datová šířka celočíselné části proměnné, chování při kvantizaci (nastává při přiřazení hodnoty s větší přesností, než je schopna cílová proměnná uložit) a chování při přetečení.

Módy kvantizace jsou: AP_TRN_ZERO (ořízne k 0) - výchozí chování, AP_RND (zaokrouhlí k plus nekonečnu), AP_RND_ZERO (zaokrouhlí k 0) a další. Módy při přetečení jsou: AP_WRAP (po přetečení hodnoty pokračují zase od 0) - výchozí chování, AP_SAT (saturace), AP_SAT_ZERO (saturace k 0) a další. Nyní si na příkladu v jazyce C++ uvedeme použití těchto typů.

```
#include "ap_int.h"  //použití celočíselných datových typů
#include "ap_fixed.h"  //použití čísel s pevnou řádovou čárkou (Fixed point)

// ap_[u]int<W> (W = 1-1024 bits)
ap_int<6> var_int1; // 6 bitová proměnná se znaménkem
ap_uint<7> var_int2; // 7 bitová proměnná bez znaménka

// ap_[u]fixed<W,I,Q,O>
ap_fixed<18,8> var_fp1; // celkem 18b (8b celá část)
ap_fixed<18,8,AP_RND_ZERO, AP_SAT> var_fp1; // kvantizace-zaokrouhlí k 0,
       přetečení-saturace
```

Zdrojový kód 2.1: Bitové datové typy.

HLS Stream Library

Streamování dat je typ přenosu, kdy jsou vzorky posílány sekvenčně za sebou. V této knihovně je třída `hls::stream`, která se chová jako nekonečná FIFO fronta, zápis a čtení je sekvenční, přečtená data nemohou být čtena podruhé. Při syntéze je implementovaná jako rozhraní `ap_fifo` s hloubkou 1. Z této třídy lze provádět blokující a neblokující čtení a zápis a kontrolu na příznaky FULL a EMPTY.

```
#include "ap_int.h"
#include "hls_stream.h"

using namespace hls;

// datová struktura typu pro rozhraní
typedef struct {
  ap_uint<2> control;
  ap_uint<18> data;
} myInterface;

void process(stream<myInterface> &in, stream<myInterface> &out){
  myInterface temp;
  bool value;
  in.read(temp); // blokující čtení
  out.write(temp); // blokující zápis
```

```
17    value = in.read_nb(temp); //neblokující čtení
18    value = out.write_nb(temp); //neblokující zápis
19  }
```

Zdrojový kód 2.2: Použití HLS stream.

HLS Math Library

HLS Math Library přidává podporu matematických funkcí pro syntézu na systémové úrovni, podobně jako standardní C Math knihovna v jazyce C. Funkce z HLS Math knihovny podporují pohyblivou nebo i některé pevnou řádovou čárkou. Matematické funkce mohou být implementovány buď pomocí IPcore (exp, 1/x, sqrt, 1/sqrt) a nebo jsou syntetizovatelné pomocí algoritmu počítající danou funkci. Některé funkce jsou přesné, některé existují ve více variantách, kde se různé implementace liší podle přesnosti. Přesnost se uvádí v jednotce ULP, která znamená, kolik nejméně významných bitů se může lišit od matematického vyjádření výsledku. 1 ULP znamená vysokou přesnost. Funkce se také mohou lišit v datových typech, které zpracovávají. Základní tvar funkce zpracovává float i double, pokud funkce končí na f, tak pouze float a některé zpracovávají i pevnou řádovou čárku (cos, sin, sqrt).

Ve zdrojovém kódu se v hlavičce může volitelně uvést `#include "hls_math.h"`. Zde jsou uvedeny některé z funkcí, které jsou podporovány v HLS Math knihovně.

Funkce	Datový typ	Přesnost	Funkce	Datový typ	Přesnost
abs	double, float	přesné	log	double, float	d: 16, f: 1
sqrt	double, float, fixed	přesné, fixed: 28-29	round	double, float	přesné
cos	double, float, fixed	10, fixed: 28-29	1/sqrt	double, float	přesné
cosf	float	1	exp	double, float	přesné
coshf	float	4	ceil	double, float	přesné
sin	double, float, fixed	10, fixed:28-29	trunc	double, float	přesné

Tabulka 2.4: Vybrané funkce z HLS Math knihovny.

HLS Video Library

HLS Video Library přidává několik datových typů a tříd usnadňujících práci se zpracováním obrazu a videa. Kromě toho přidává tato knihovna podporu pro používání OpenCV funkcí. Jelikož celá OpenCV aplikace nemůže být syntetizovatelná, ale jen vybrané funkce (například filtry obrazu), existuje zde rozhraní mezi OpenCV aplikací běžící na procesoru a syntetizovatelnou funkcí, která je akcelerována v FPGA. Pro použití knihovny je potřeba do hlavičkového souboru napsat `#include "hls_video.h"`.

Knihovna přidává podporu datových typů, které se používají pro reprezentaci pixelů. Zde je uvedeno několik z nich a v závorce je uveden význam jednotlivých bytů v datovém typu: `rgb_8 (B0: G, B1: B, B2: R)`, `rgba_8 (B0: G, B1: B, B2: R, B3: A)`, `yuv422_8 (B0: Y, B1: UV)`, `bayer_8 (B0: RGB)`.

Kromě datových typů zde najdeme také třídy Memory Line Buffer a Memory Window Buffer. Obě třídy pracují s různými datovými typy a lze rozdělovat tyto vyrovnávací paměti do více bank, aby se zvýšila propustnost čtení a zápisu. U třídy Line Buffer představuje každý sloupec jednotlivou paměť. Lze z něj číst, vkládat na konec nebo začátek a nebo posouvat nahoru nebo dolů. Třída Window Buffer představuje 2D matici bodů, ke kterým

můžeme libovolně přistupovat pomocí indexů řádku a sloupce. Rovněž můžeme celou matici posunout do 4 směrů (nahoru, dolů, doprava, doleva) pomocí jedné metody.

```
#include "hls_video.h"
using namespace hls;

LineBuffer<char,3,4> lb1; // 3 řádky, 4 sloupce, datový typ char
lb1.insert_top(10,0); // do 0. sloupce vloží hodnotu 10
lb1.shift_down(0); // sloupec 0 posune dolů
char val = lb1.getval(0,1); // vložená hodnota 10 bude nyní na řádku 1

Window<char,3,4> w1; // 3 řádky, 4 sloupce, datový typ char
w1.insert(10, 1, 2); // vloží hodnotu 10 na řádek 1, sloupec 2
w1.shift_left(); // hodnota 10 je na řádku 1, sloupec 1
w1.shift_up(); // hodnota 10 je na řádku 0, sloupec 1
```

Zdrojový kód 2.3: Použití HLS Video Library.

Další funkcí knihovny je vytvoření rozhraní mezi OpenCV aplikací v procesoru a akcelerovanou funkcí v FPGA. Pro použití této knihovny je potřeba použít knihovnu `hls_opencv.h`. Funkce v FPGA přijímá přes AXI-Stream data a interně je převádí na `hls::Mat`. Poté je možné zavolat funkce OpenCV, které jsou upravené pro syntézu. Jsou to funkce pro práci s maticemi (AbsDiff, Avg, Max, Min, Mean, Mul, Sum, Scale, Threshold, Zero, ..) a také jednodušší filtry obrazu (CornerHarris, Dilate, EqualizeHist, Erode, FASTX, Filter2D, GaussianBlur, Harris, HoughLines2, PaintMask, Sobel, ...). Následuje příklad použití funkce sobelova filtru za použití OpenCV funkce.

```
#include "hls_video.h"
using namespace hls;
typedef stream<ap_axiu<32,1,1,1>> AXISTREAM;

void sobel_filter(AXISTREAM &IN, AXISTREAM &OUT){
  Mat<HEIGHT, WIDTH, HLS_8UC3> src_img, dest_img; // deklarace hls::Mat

  AXIvideo2Mat(IN, src_img); // převede AXI Stream do hls::Mat
  Sobel<1,0,3>(src_img, dest_img); // zavolá funkci z OpenCV
  Mat2AXIvideo(dest_img, OUT); // převede hls::Mat do AXI Streamu
}
```

Zdrojový kód 2.4: Použití HLS Video Library.

HLS IP Library

Tato knihovna umožňuje implementovat do FPGA v jazyce C++ některé z IP bloků. Jedná se o tři bloky: FFT, FIR Filter a Shift register.

Pro použití FFT je potřeba použít knihovnu `hls_fft.h`. Pro IP blok musíme nastavit parametry v předdefinované struktuře `hls::ip_fft::params_t`, poté je třeba nastavit konfiguraci za běhu a spustit funkci.

Použití FIR filtru vyžaduje knihovnu `hls_fir.h`. Filtr rovněž vyžaduje konfiguraci pomocí struktury `hls::ip_fir::params_t` a poté spuštění samotného filtru.

Shift Register v knihovně `ap_shift_reg.h` je mapován přímo do SRL prvků FPGA. Z posuvného registru se může číst z jakékoli adresy, ale zapisovat pouze na začátek. Následuje ukázka použití posuvného registru.

```
#include "ap_shift_reg.h"
static ap_shift_reg<int, 3> sreg; // 3 prvky typu int, sreg musí být static
int val = sreg.shift(10, 2, true); // uloží na pozici 0 hodnotu 10 a pokud
    je true, posune vše o 1 doprava. Do hodnoty val uloží prvek z indexu 2.
```

Zdrojový kód 2.5: Použití Posuvného registru v třídě ap_shift_reg.

HLS Linear Algebra Library

Knihovna HLS Linear Algebra má hlavičkový soubor v `hls_linear_algebra.h`. Poskytuje podporu pro obecné funkce používané v lineární algebře. Pracuje s dvourozměrnými maticemi. Knihovna podporuje tyto funkce: cholesky, cholesky_inverse, matrix_multiply, qrf, qr_inverse a svd. Funkce podporují datový typ float, některé i pevnou řádovou čárku.

Rozhraní komponenty

Každá komponenta může mít obecně více druhů rozhraní. Rozhraní se definuje pomocí direktivy `#pragma HLS interface <mode> port=<variable name>`. Rozhraní definujeme k top funkci, k parametrům funkce nebo ke globální proměnné. Pokud definujeme rozhraní top funkce, uvedeme místo názvu proměnné klíčové slovo return.

- **ap_ctrl_none** - mód k top-funkci, nevytváří žádné další signály, nevytváří handshake protokol. Komponenta spotřebuje méně zdrojů, protože zde není další řídící logika funkce.
- **ap_ctrl_hs** - výchozí mód top funkce. Implementuje handshake protokol. Vytváří signály: IN: `ap_start` - musí být v logické 1, pokud chceme spustit funkci, OUT: `ap_done` - signalizuje, že funkce skončila a že návratová hodnota je validní, `ap_idle` - indikuje, že funkce je nečinná, `ap_ready` - značí, že funkce je připravená přijímat nová data (u pipeline).
- **ap_ctrl_chain** - mód k top funkci, podobný předchozímu `ap_ctrl_hs`, používá se pro zřetězené bloky, které zpracovávají streamovaná data. Přidává zde ještě signál `ap_continue` - vstupní proměnná, která signalizuje, že další blok v lince nemůže přijímat nová data.
- **ap_none** - mód pro parametry funkce a globální proměnné. K datovým signálům se nevytváří žádné další signály. Výchozí mód u vstupních proměnných (Read-only).
- **ap_stable** - tento mód indikuje, že proměnná se nezmění od resetu do dalšího resetu. Pouze pro vstupní proměnné.
- **ap_vld** - výchozí mód u výstupních proměnných (Write-only). Je vytvořen další signál `<portName>_vld`, který indikuje, že proměnná je validní.
- **ap_ovld** - výchozí mód u vstupně - výstupních proměnných (Read-Write). Vstupní signály implementovány jako `ap_none`, výstupní jako `ap_vld`.
- **ap_ack** - Vytvoří další signál `<portName>_ack`, který při vstupních proměnných značí druhé komponentě, že z proměnné bylo přečteno. U výstupních proměnných je pozastaven zápis, dokud není potvrzeno přečtení.

- **ap_hs** - jedná se o kombinaci módů `ap_vld` a `ap_ack`, jsou vytvořeny oba signály `<portName>_vld` a `<portName>_ack`. Vstupní proměnné jsou čteny, až je signál validní a do výstupních proměnných je zapisováno, až je potvrzeno přečtení předchozí hodnoty.
- **ap_fifo** - implementuje ukazatel na pole, které se chová jako FIFO paměť. Při čtení se generuje signál read a příznak neprázdnosti. Při zápisu se generuje signál write a příznak, že fronta není plná.
- **ap_memory** - implementuje proměnnou typu pole pro přístup do externí paměti RAM. Vytvoří další signály jako CE, WE. Typy signálů jsou určeny podle typu paměti, ke které se připojuje.
- **bram** - vytvoří jeden port, který může být připojen k Xilinx BRAM komponentě.
- **ap_bus** - implementuje ukazatel na rozhraní sběrnice. Pro čtení a zápis jsou generovány řídící signály podporující standardní FIFO sběrnici. Obsahuje vnitřní vyrovnávací paměť a pracuje v dávkovém režimu.
- **axis** - implementuje AXI4-Stream protokol. AXI4-Stream obsahuje datový a několik řídících signálů. Zde je ukázána struktura protokolu AXI4-Stream.

```
template<int D,int U,int TI,int TD>
struct ap_axis{
  ap_int<D> data; // datový signál
  ap_uint<D/8> keep; // značí, že přijde zbytek dat
  ap_uint<D/8> strb; // používá se ke kódování
  ap_uint<U> user;
  ap_uint<1> last; // indikuje poslední data v paketu
  ap_uint<TI> id;
  ap_uint<TD> dest;
};
```

Zdrojový kód 2.6: Struktura AXI4-Stream.

- **s_axilite** - implementuje AXI4-Lite slave protokol. Zde lze seskupovat více portů do jednoho AXI4-Lite rozhraní.
- **m_axi** - implementuje AXI4 master protokol.

Seznam direktiv pro syntézu

V prostředí Xilinx Vivado HLS lze vkládat do zdrojového kódu nebo do tcl skriptu následující direktivy [9].

- **allocation** - definuje limity nebo počet RTL instancí, které budou použity při implementaci konkrétní funkce nebo operátoru.
- **array_map** - mapuje více menších polí do jednoho většího.
- **array_partition** - aplikuje optimalizace rozdělení polí popsanou v kapitole 2.1.
- **array_reshape** - kombinuje array_map a array_partition, vytvoří jedno velké pole a automaticky přeskládá vertikálně prvky pole do nového s větší šířkou slova.

- **clock** - definuje hodiny ke konkrétní funkci. Podpora pouze u SystemC jazyka. C/C++ podporují pouze jedny hodiny.
- **dataflow** - optimalizace rychlosti pro funkce nebo cykly. Automaticky se analyzují a plánují operace tak, aby se minimalizovala latence.
- **data_pack** - seskupuje datovou strukturu do jedné skalární hodnoty s větší datovou šířkou.
- **dependence** - pomocí této direktivy můžeme povolit nebo zakázat datovou závislost. Jedná se o klasické závislosti typu RAW, WAR, WAW. Můžeme tak například odstranit falešnou závislost u přístupu k polí v iteracích smyčky nebo ve zřetězené lince.
- **expression_balance** - optimalizace kódu, která analyzuje operace a snaží se je optimalizovaně naplánovat. Ve výchozím nastavení je zapnuto, pomocí této direktivy lze vypnout.
- **function_instantiate** - umožňuje vytvořit vlastní RTL implementaci ke každé instanci funkce. Každá instance může být optimalizovaná zvlášť.
- **inline** - aplikuje optimalizace inline popsanou v kapitole 2.1.
- **interface** - nastaví rozhraní komponenty, možnosti rozhraní jsou popsány v kapitole 2.3.
- **latency** - nastaví maximální a minimální latenci funkce, cyklu nebo bloku.
- **loop_flatten** - aplikuje optimalizace flattening popsanou v kapitole 2.1.
- **loop_merge** -aplikuje optimalizace slučování smyček popsanou v kapitole 2.1.
- **loop_tripcount** - můžeme nastavit minimální, průměrný a maximální počet iterací, který se provede v cyklu. Umožňuje lepší analýzu návrhu.
- **occurrence** - aplikuje se pokud nějaký blok (funkce, cyklus) je spouštěn méně krát, než zbytek zřetězené funkce.
- **pipeline** - aplikuje optimalizaci zřetězení popsanou v kapitole 2.1.
- **protocol** - umožňuje manuálně specifikovat vlastní protokol, HLS zde nevloží hodinové takty, ty jsou explicitně určeny pomocí příkazů wait().
- **reset** - přidá reset ke globální nebo statické proměnné.
- **resource** - umožňuje určit RTL prvek, do kterého bude proměnná nebo operace implementována (např. určení jedno nebo dvou portové BRAM u pole, typ násobičky, ...).
- **stream** - umožňuje z pole vytvořit stream, kde se používá FIFO místo RAM paměti.
- **top** - definuje název top funkce.
- **unroll** - aplikuje optimalizaci rozbalení smyček popsanou v kapitole 2.1.

Kapitola 3

Návrh aplikace pro zpracování videa

Kapitola se věnuje popisu návrhu aplikace pro zpracování videa na platformě Xilinx ZYNQ. První část kapitoly popisuje specifikaci zadání aplikace. Problém je pak dekomponován na jednodušší bloky. Celková aplikace je rozdělena mezi softwarovou část na procesoru ARM a hardwarovou část na FPGA. Aplikace obsahuje obrazové filtry a klasifikátor SPZ (Státní poznávací značka). Některé části aplikace jsou převzaté. Hlavní částí aplikace jsou komponenty v FPGA, které budou implementovány za použití syntézy na systémové úrovni v prostředí Xilinx Vivado HLS.

3.1 Specifikace zadání

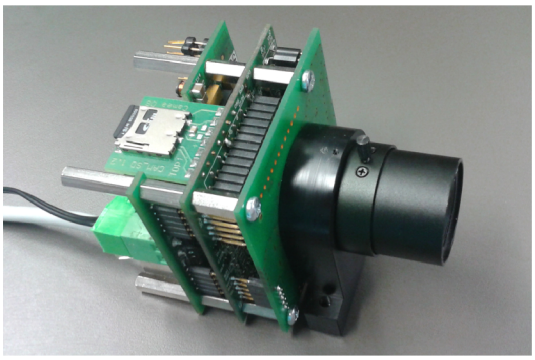
Aplikace bude přijímat obraz z kamery nebo ze statického obrázku a dále jej zpracuje v FPGA. Zpracování videa probíhá v reálném čase. Aplikace pracuje s HD (High Definition) rozlišením (1280 * 720 px). Výsledek se zobrazí na HDMI monitoru. V FPGA budou implementovány obrazové filtry. Sobelův filtr pro detekci hran, mediánový filtr pro odstranění šumu a bilaterální filtr pro vyhlazení obrazu. Kromě filtrů aplikace obsahuje AdaBoost klasifikátor pro detekci SPZ. Detektor vyhledá v obraze jednu SPZ a po nalezení vykreslí rámeček kolem nalezeného objektu. Konfiguraci aplikace lze provádět za běhu přes skripty spuštěnými pod OS Linux na ARM procesoru.

Technické vybavení k aplikaci

Aplikace bude spuštěna na vývojové desce Xilinx ZC702, která je popsána v kapitole 2.2. Ke kitu se připojí přes rozhraní Ethernet kamera Unicam M621 z firmy CAMEA. Kamera je zobrazena na obrázku 3.1, pracuje v 8 bitovém režimu ve stupních šedi. Rozlišení kamery je 752 * 478 pixelů a frekvence 60 snímků/s a používá UDP protokol [4]. Jelikož aplikace pracuje s HD rozlišením (1280 * 720 px), proto bude při použití kamery obraz vyplněn pozadím.

Převzaté části

V rámci vytváření této aplikace jsem spolupracoval s Ing. Petrem Musilem a Ing. Martinem Musilem, od kterých jsem obdržel zprovozněný kit s nainstalovaným systémem Linux.



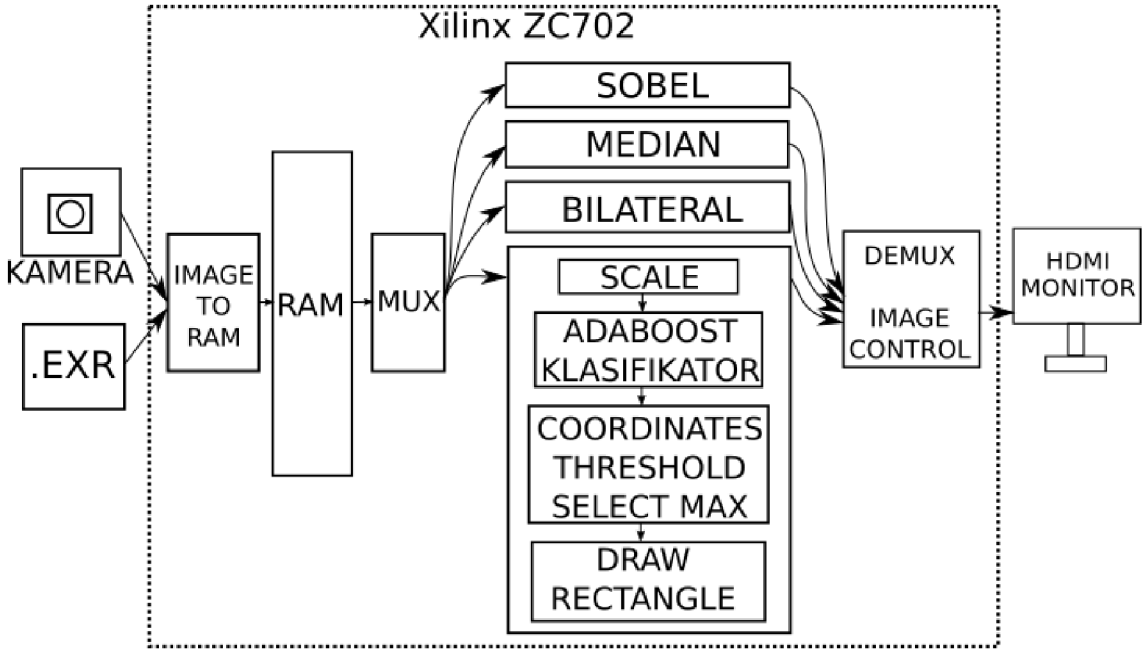
Obrázek 3.1: kamera Unicam M621

Rovněž bylo vyřešeno získávání obrazu z paměti RAM a obsluha HDMI monitoru. Při implementaci bilaterálního filtru jsem také obdržel SW řešení algoritmu pro běžné CPU.

Při implementaci architektury detektoru registračních značek vozidel jsem spolupracoval s Ing. Filipem Kadlčkem z firmy CAMEA. Získal jsem natrénovaný klasifikátor AdaBoost na SPZ [3]. Také jsem obdržel popis architektury, která bude implementována pomocí prostředí Vivado HLS.

3.2 Návrh řešení aplikace

Jelikož platforma Xilinx ZYNQ obsahuje ARM procesor i FPGA, je vhodné aplikaci dekomponovat na softwarovou a hardwarovou část. Také je potřeba navrhnout rozhraní mezi jednotlivými bloky.



Obrázek 3.2: Schéma aplikace pro zpracování obrazu.

Na obrázku 3.2 vidíme celkové schéma aplikace. Vstupní obraz je získáván buď z kamery

nebo z obrázku. Obrázek může být uložen v libovolném formátu (.exr, .jpg, .png, ...) a v libovolném rozlišení. Aplikace tento obrázek zpracuje a uloží do předem definované oblasti v paměti. DMA přenos z FPGA načítá z této oblasti data pro proud videa. V paměti je každý pixel uložen ve stupních šedi na 32 bitů. Aplikace pro uložení obrázku umí nastavovat bitovou šířku pixelů a zbytek doplnit nulami. Obrazové filtry v aplikaci pracují s 18-ti bitovými pixely.

Vstupní data také mohou být získávána z kamery Unicam M621, která je popsaná v kapitole 3.1. Pro obsluhu příjmu dat z Ethernetového rozhraní a ukládání do paměti bude vytvořena aplikace pro ARM procesor. Jelikož data přichází přes UDP protokol nezarovnaná, jsou dále zpracována do matice bodů. Výsledek je rozšířen na 18 bitů a ukládá se do paměti RAM.

V FPGA části je použita komponenta od firmy Xilinx, která pomocí DMA přenosu načítá v dávkách data z RAM a vytváří z nich video AXI-Stream. Protokol AXI-Stream je popsán v podkapitole 3.2. Data poté vstupují do komponent vytvořených pomocí syntézy na systémové úrovni. Komponenta MUX, která slouží jako multiplexor výběru filtru, převádí 32 bitový signál na 18 bitový, se kterým pracují samotné filtry obrazu. Komponenta MUX a současně i demultiplexor DEMUX jsou ovládány uživatelem přes protokol AXI4-Lite. Uživatel tak může přepínat za běhu mezi jednotlivými filtry. Jednotlivé filtry obrazu a detektor SPZ jsou popsány v následujících kapitolách. Výstup z filtrů nebo detektoru lze ještě upravit pomocí komponenty na úpravu jasu, která vynásobí data desetinnou hodnotou. Rovněž ji lze konfigurovat za běhu pomocí protokolu AXI4-Lite. Na konci zpracování komponenta upraví signál z 18 bitů na 10 bitů, který je podporován řadičem HDMI. Dále jsou data zpracována HDMI kontrolérem a zobrazena na monitoru.

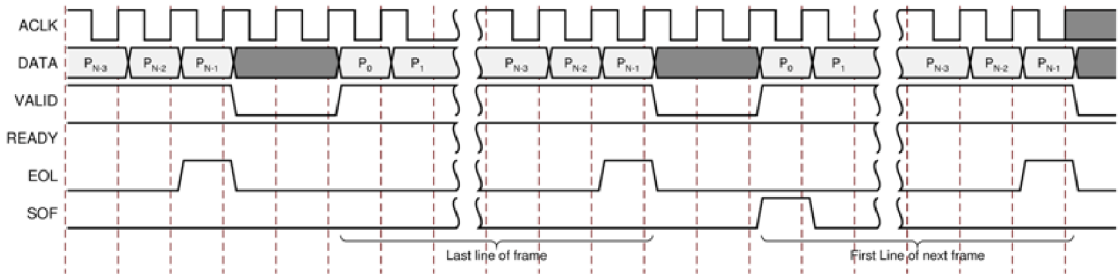
Přenos a zobrazení videa

K přenosu videa v FPGA lze využít protokol AXI-Stream. AXI-Stream je vhodný pro vysokorychlostní přenosy proudových dat. Posílá data v dávkách a zajišťuje Handshake protokol. AXI-Stream lze obecně použít pro obecný přenos dat. Protokol obsahuje několik signálů. Signál DATA může mít 1-1024 bitů datovou šířku a je povinný. Další signály jsou pro řízení Handshake protokolu - TVALID a TREADY. TVALID je v logické 1, pokud výstupní data jsou platná. Signál TREADY je v logické 1, pokud je komponenta připravena přijmout nová data. Další signály jsou volitelné a jsou to: TSTRB, TKEEP, TLAST, TID, TDEST, a TUSER.

Pro proud videa se používá video AXI-Stream. Protokol je popsán v dokumentu UG934 [8]. V protokolu probíhá přenos dat po pixelech. Kromě signálu TDATA s hodnotou pixelu se ještě přenáší signál začátku snímku (SOF - Start Of Frame) a signál konce řádku (EOL - End Of Line). Signál SOF je přenášen v TUSER signále a EOL je přenášen v TLAST signále. Na obrázku 3.3 lze vidět příklad přenosu snímku pomocí video AXI-Strem protokolu.

Konfigurace FPGA komponent z OS Linux

AXI-Lite protokol je použit v aplikaci pro konfiguraci FPGA komponent za běhu z OS Linux. AXI-Lite protokol se používá pro nízko rychlostní přenosy. Registry při použití AXI-Lite protokolu jsou adresovatelné. Adresy jsou přiděleny z adresového prostoru nad poslední adresou paměti RAM. Pak lze z operačního systému přistupovat k těmto registrům v FPGA. AXI-Lite protokol obsahuje několik signálů pro řízení. Signály jsou pro čtení a zápis adresy (AWADDR, AWVALID, AWREADY) a pro čtení a zápis dat (WDATA, WSTRB, WVALID, WREADY) a kanál, který odpovídá má signály s písmenem B (BVALID, BREADY).



Obrázek 3.3: Video AXI-Stream protokol. Převzato z [8]

.

Analogicky existují ještě signály pro čtení, kde v označení signálu je místo W písmeno R [12].

Aplikace v ARM procesoru pro vstupní video

Aplikace je rozdělena na část spuštěnou v operačním systému Linux na procesoru ARM a část v FPGA. Jelikož je rozhraní Ethernet připojeno k procesoru ARM, bude zde spuštěna aplikace pro zpracování dat z kamery. Přijímá se UDP protokol s daty a kontrolními informacemi. Aplikace má dvě vlákna. První přijímá data z rozhraní, druhé vlákno hodnoty rozšiřuje na 18b a ukládá do definované oblasti v paměti RAM. Aplikace v ARM využívá funkce z knihovny OpenCV. Pro rychlé ukládání dat do paměti RAM je použit driver v jádru OS Linux, který implementoval Ing. Martin Musil. Používá se voláním funkce ioctl() a write() a dosahuje vyšší efektivity než funkce mmap().

Komponenty pro řízení videa

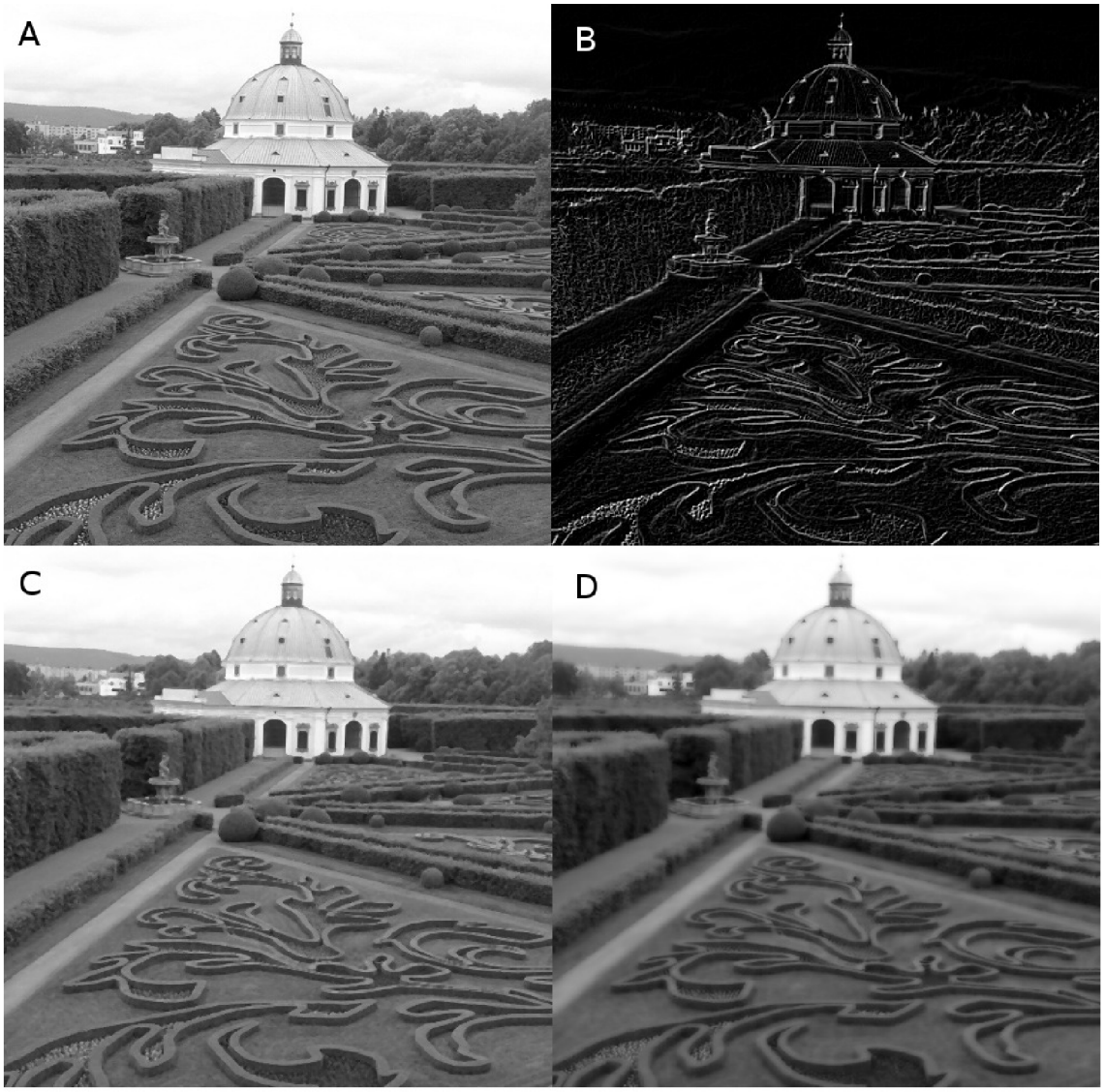
V FPGA je umístěna DMA komponenta, která vyčítá z paměti RAM data a převádí je na video AXI-Stream. K datům jsou také přidány signály značící začátek snímku (Start Of Frame) a signál konce řádku. Nyní budou popsány jednotlivé komponenty, které budou implementovány v prostředí Vivado HLS. Jedná se celkem o 12 komponent s různou složitostí.

Proud videa z DMA je přiveden do komponenty AxiMultiplexor. Tato komponenta přepíná mezi jednotlivými filtry. Je řízena pomocí AXI-Lite protokolu. Filtry lze přepínat pomocí skriptu v prostředí Linux. Analogickou komponentou je AxiDemultiplexor, která je umístěna v návrhu po aplikování filtru a předává video signál pro zobrazení.

Komponenta AxiVideoBrightness nastavuje jas videa, hodnota jasu je opět řízena pomocí protokolu AXI-Lite skriptem z OS Linux. Před výstupem do HDMI jádra se upravuje bitová šířka dat z 18b na 10b a také se provádí ořez krajních hodnot, kde jsou řídicí data HDMI. To provádí komponenta AxiVideo2HDMI.

Komponenty obrazových filtrů

Při praktickém použití kamery v úlohách počítačového vidění nebo ostatních průmyslových úlohách, je potřeba zpravidla vstupní obraz předzpracovat. K tomu se používají digitální filtry, které se aplikují na obraz. Existuje mnoho filtrů s celou řadou uplatnění. Například filtry pro zostření, rozmazání, úpravu histogramu, detekci hran, detekce významných bodů obrazu nebo detekce jiných zájmových oblastí.



Obrázek 3.4: Ukázka filtrů: A - nefiltrovaný obrázek, B - sobelův filtr, C - mediánový filtr, D - bilaterální filtr.

- **Sobelův filtr** - slouží jako detektor hran. Na obraz je aplikována konvoluce s maskou Sobelova filtru o velikosti 3x3 pixely. Maska se aplikuje dvakrát, poprvé detekuje hrany ve směru osy x a poté ve směru osy y. Poté je proveden výpočet na spojení do obou směrů. Filtr také ošetřuje okrajové hodnoty a používá nejbližší platnou hodnotu.

- **Mediánový filtr** - filtr se používá k odstranění náhodného šumu. Pracuje s filtrovacím oknem 5x5 pixelů. Porovnávají se sousední hodnoty a medián se spočítá po 24 krocích.

- **Bilaterální filtr** - nahrazuje hodnotu pixelu průměrnou hodnotou okolních podobných pixelů. Tudíž dochází k rozmazání obrazu, ale zároveň jsou zachovány ostré

přechody [1]. V projektu se používá velikost okna 11x11 pixelů. Pokud je velký rozdíl mezi pixely, tak hranu zanechá. Postupně pro menší změny v obraze výsledný obraz vyhlazuje.

Při implementaci filtrů na běžném CPU provádíme filtraci klasickým způsobem, kdy načteme pixely obrázku do matice a přes vnořený cyklus procházíme jednotlivé sloupce a řádky matice. Při zpracování obrazu v FPGA ale musíme algoritmus upravit. Pixely totiž přicházejí přes protokol AXI4-Stream a každý takt dostáváme nový pixel. Zároveň očekáváme, že každý takt bude komponenta produkovat pixel na výstup. Načtení celého obrázku, provedení filtrace a následné odesílání po pixelech není možné z důvodu omezené paměti, která nedostačuje na uložení celého obrázku v HD rozlišení.

Řešením je použití menší vyrovnávací paměti, která by si pamatovala jen počet řádků obrazu, které jsou potřeba pro filtrovací masku a fungovala by jako FIFO paměť. Schéma použití FIFO lze vidět na obrázku 3.5. Každým taktem se okno a celá FIFO paměť posouvá o jednu hodnotu. Hodnoty, které jsou zrovna v okně, jsou uloženy v registrech (jedná se například jen o 9 hodnot u okna 3x3 pixelů), protože je potřeba naráz přistoupit ke všem hodnotám. Aby se ušetřila plocha na čipu, tak zbytek řádku je uložen v dvouportové BRAM paměti. Každý takt je jedním portem proveden zápis a druhým portem se přečte hodnota FIFO fronty.

Obrázek 3.5: Filtrace obrazu v FPGA. Počítá se červený pixel, potřebuje k tomu okolí 3x3 pixelů. Žlutý bod je nový v lince, oranžový se zahazuje po skončení výpočtu. Maska 3x3 je v registrech, modrý zbytek řádku je v paměti BRAM.

Funkce filtru bude zřetězena s inicializačním intervalem jeden hodinový takt. Funkce bude obsahovat čítač řádků a sloupců obrazu, aby komponenta správně dávala na výstup signály End Of Line a Start Of Frame. Komponenta totiž bude přidávat určitou latenci při zpracovávání.

Komponenty architektury AdaBoost klasifikátoru

Architektura klasifikátoru je určena pro nalezení registrační značky vozidla v obraze. V kapitole 3.1 je zmíněno, že samotný klasifikátor AdaBoost je převzatý a již implementovaný v jazyce VHDL. Cílem bylo vytvořit architekturu kolem klasifikátoru. Klasifikátor má na vstupu 8 bitovou hodnotu obrazu a na výstupu dává hodnotu pravděpodobnosti, zda daný pixel odpovídá hledanému objektu. Samotný klasifikátor má definovanou latenci, se kterou se počítá při získávání výstupu.

Kvůli ušetření zdrojů je použit algoritmus AdaBoost s 50 slabými klasifikátory. Detektor používá slabé příznaky LBP (Local Binary Pattern). Jelikož samotný klasifikátor používá rozhraní datových signálů, je nutné vytvořit kolem klasifikátoru komponenty s tímto rozhraním.

Nejprve je video AXI-Stream převedeno na signály pomocí komponenty Axis2Wire. Poté následuje jednotka měnící rozlišení obrazu o pětinu. Komponenta se jmenuje ImageScale.

[1] homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

Změna rozlišení se provádí metodou nejbližšího souseda. Z této komponenty vystupuje video do klasifikátoru. Následující tabulka 3.1 ukazuje jednotlivé stupně zmenšení obrazu a velikost SPZ, která je detekována při daném zmenšení.

Rozlišení obrazu	Velikost SPZ	Rozlišení obrazu	Velikost SPZ
1280 x 720	60 x 15	268 x 151	286 x 89
1024 x 576	75 x 19	214 x 121	357 x 112
819 x 561	93 x 23	171 x 97	447 x 140
655 x 369	117 x 37	136 x 77	558 x 175
524 x 295	146 x 46	108 x 62	698 x 218
419 x 236	183 x 57	86 x 49	873 x 273
335 x 189	228 x 72	68 x 40	1091 x 341

Tabulka 3.1: Velikost detekované SPZ na základě zmenšení obrázku.

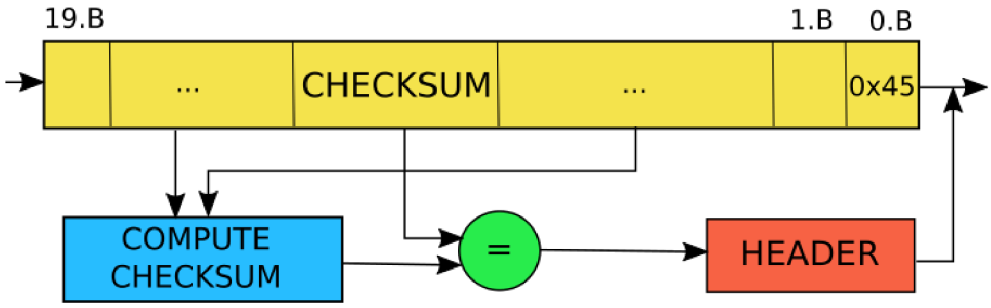
Na výstupu klasifikátoru je komponenta ClasifOut, která počítá aktuální řádek a sloupec. Tuto hodnotu počítá na základě signálu Start Of Frame. Také provádí prahování výstupu z klasifikátoru. Pokud je hodnota větší jak práh, posílá ji do komponenty ClasifMax. Zde se hledá maximální odezva z celého snímku. Jednou za snímek se pošle maximum do komponenty, která vykreslí rámeček na daných souřadnicích - komponenta DrawBorder. Aby se nemusel ukládat celý snímek do kterého se kreslí rámeček, tak se kreslí do následujícího snímku. U videa toto opatření nevadí a zároveň dojde k ušetření zdrojů.

Rozšíření diplomové práce - Komponenta pro hledání začátku paketu

Komponenta není zahrnuta do aplikace pro zpracování obrazu, ale byla vytvořena v rámci výzkumu na Fakultě informačních technologií. Komponenta má za cíl vyhledat v proudu bytů začátek paketu v síťovém provozu a v případě nalezení do proudu vložit vlastně definovanou hlavičku a poté přeposlat celý paket. Vložená hlavička do proudu může obsahovat zdrojovou nebo cílovou IP adresu, typ paketu nebo další informace. Tato vlastní pevně definovaná hlavička se vkládá kvůli dalšímu zpracování nebo filtrování paketů.

Hledání začátku paketu funguje pro IPv4 hlavičky, kdy se hledá počáteční byte 0x45 a poté se spočítá kontrolní součet (chcecksum) hlavičky. Pokud obě podmínky odpovídají začátku paketu, tak se do proudu vloží hlavička. Při posílání hlavičky je pozastaveno čtení nových dat do zřetězené linky. Na obrázku 3.6 lze vidět celkové schéma komponenty.

Poslední byte paketu obsahuje signál LAST. Když přišel takový paket a na vstupu nepřicházejí další data, tak je potřeba postupně vysunout zbytek paketu na výstup komponenty.



Obrázek 3.6: Schéma komponenty pro hledání začátku paketu.

Kapitola 4

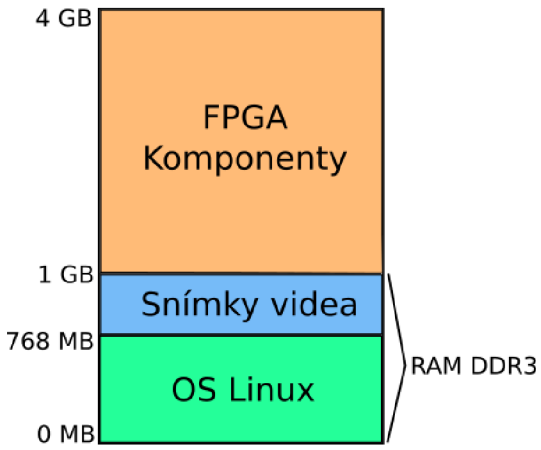
Implementace a testování aplikace

V této kapitole bude popsána implementace jednotlivých komponent v prostředí Xilinx Vivado HLS ve verzi 2014.4. Ke každé komponentě je tabulka, kolik spotřebuje zdrojů. V kapitole je také popsán paměťový model aplikace a jak byla řešena komunikace mezi ARM procesorem a FPGA. Další část kapitoly se věnuje porovnání implementace pomocí HLS s implementací v HDL jazycích. Poslední oddíl popisuje, jak probíhalo testování aplikace.

4.1 Implementace v prostředí Xilinx Vivado

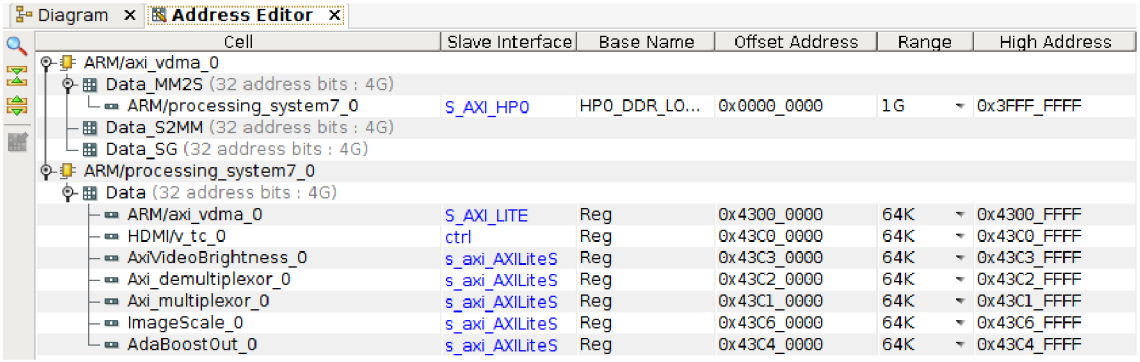
Paměťový model aplikace

Na obrázku 4.1 lze vidět graficky jednotlivé části paměti. Vývojová deska Xilinx ZC702 obsahuje 1GB DDR3 operační paměti. Část paměti (768 MB) je určena pro běh operačního systému Linux. A část (256 MB) je určena pro uložení snímku videa. Paměti jsou oddělené, protože v FPGA je DMA komponenta, která vyčítá snímky z předem definované adresy. Pokud bychom paměti neoddělili, operační systém si může v tomto místě alokovat paměť pro spuštěné aplikace a bude přepisovat snímky videa jinými daty. Jelikož u platfromy ZYNQ se jedná o 32b architekturu, lze adresovat do 4GB. Adresy, které jsou vyšší, jak 0x3FFFFFFF nesměřují do paměti RAM, ale slouží k adresování komponent v FPGA. Jedná se například o AXI-Lite protokol, který přiděluje proměnným v FPGA adresu. Rovněž lze tak konfigurovat komponenty od výrobce, které mají adresovatelné registry.



Obrázek 4.1: Paměťový model aplikace.

V prostředí Xilinx Vivado je editor adres, ve kterém lze vidět přiřazené adresy ke komponentám, které jsou v návrhu. Příklad přidělení adres lze vidět na obrázku 4.2. Pro zapisování dat na určitou adresu lze použít aplikaci `devmem2` v prostředí Linux. Tato aplikace zapisuje na fyzickou adresu v paměti.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
ARM/axi_vdma_0					
Data_MM2S (32 address bits : 4G)					
ARM/processing_system7_0	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
Data_SG (32 address bits : 4G)					
ARM/processing_system7_0					
Data (32 address bits : 4G)					
ARM/axi_vdma_0	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
HDMI/v_tc_0	ctrl	Reg	0x43C0_0000	64K	0x43C0_FFFF
AxiVideoBrightness_0	s_axi_AXILiteS	Reg	0x43C3_0000	64K	0x43C3_FFFF
Axi_demultiplexor_0	s_axi_AXILiteS	Reg	0x43C2_0000	64K	0x43C2_FFFF
Axi_multiplexor_0	s_axi_AXILiteS	Reg	0x43C1_0000	64K	0x43C1_FFFF
ImageScale_0	s_axi_AXILiteS	Reg	0x43C6_0000	64K	0x43C6_FFFF
AdaBoostOut_0	s_axi_AXILiteS	Reg	0x43C4_0000	64K	0x43C4_FFFF

Obrázek 4.2: Editor adres v prostředí Xilinx Vivado.

Přístup do fyzické paměti v OS Linux

V aplikaci je také řešen přístup do fyzické paměti z uživatelské aplikace spuštěné na OS Linux. Jádro operačního systému poskytuje volání `mmap` (memory map) [1]. Volání `mmap` dokáže mapovat soubory nebo zařízení do virtuální paměti volajícího procesu. Tak lze mapovat fyzickou paměť na zadané adrese do virtuální paměti procesu. Předem ovšem musí být zajištěno, že operační systém v této oblasti paměti nemůže alokovat data pro spuštěné procesy. V následujícím příkladu je ukázka volání `mmap`. V této ukázce dostáváme virtuální adresu pro uložení snímku v HD rozlišení, který bude uložen na adrese 0x3E000000. Každý pixel je uložen na 32 bitů.

```
1  const int FRAME_WIDTH = 1280;
2  const int FRAME_HEIGHT = 720;
3  const int FRAME_DATA_SIZE = FRAME_WIDTH*FRAME_HEIGHT*4; //velikost paměti
4  const int FRAME_DATA_ADDRESS  = 0x3E000000; //fyzická adresa
5
6  int32_t *virtual_addr = NULL;
7  //otevření zařízení /dev/mem
8  int fdmem = open( "/dev/mem", O_RDWR | O_SYNC );
9  //volání mmap - přidělení virtuální adresy z fyzické paměti na offsetu
       0x3E000000
10 virtual_addr = (int32_t *)(mmap(0, FRAME_DATA_SIZE, PROT_READ|PROT_WRITE,
       MAP_SHARED, fdmem, FRAME_DATA_ADDRESS));
11 //zápis 1 na nultý pixel obrázku
12 virtual_addr[0] = 1;
```

Zdrojový kód 4.1: Příklad volání mmap.

Aplikace pro zpracování dat z kamery

Aplikace pro zpracování dat z kamery je psaná v jazyce C++ a je spuštěná pod OS Linux. Využívá dvě vlákna. Jedno vlákno přijímá UDP pakety z kamery. Podle ID paketu zjišťuje

[1] http://man7.org/linux/man-pages/man2/mmap.2.html

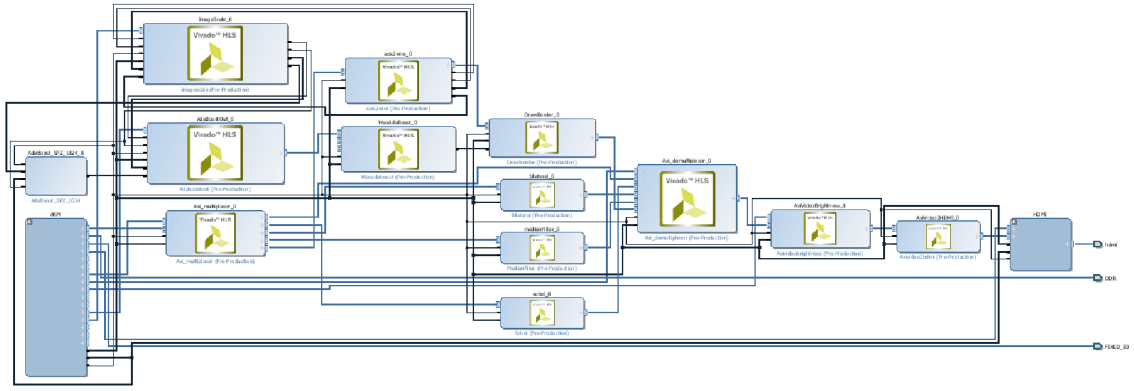
souřadnice dat ve snímku. Vzniká tak postupně matice bodů odpovídající snímku. Na konci snímku přichází kontrolní paket označující tuto událost. Po naplnění celého snímku daty je spuštěno druhé vlákno, které ukládá data do fyzické paměti.

Přístup do fyzické paměti pomocí mmap je použit v aplikaci pro přenos snímku ze statického obrázku nebo při zapisování do registrů v FPGA. Pro přenos videa z kamery je však výkon nedostačující. Proto Ing. Martin Musil vytvořil driver v jádře OS Linux, který přímo zapisuje do fyzické paměti. K driveru se přistupuje voláním jádra `ioctl` [2] a `write`. Pomocí `ioctl` se nastavuje offset, kam zapisovat a pomocí `write` se zapíše blok dat přímo do fyzické paměti.

Spuštěná aplikace spotřebovává asi 70% CPU (z 200%, protože se jedná o 2 jádrový procesor) na ARM Cortex A9.

Vytvoření schématu aplikace

V kapitole 2.3 bylo popsáno, jak se vytváří komponenta ve vývojovém prostředí Xilinx Vivado HLS. Následně je však nutné sestavit z komponent celkovou aplikaci. Použité komponenty mohou být z katalogu od výrobce nebo napsané v HDL jazycích nebo například vytvořené pomocí Vivado HLS. Na obrázku 4.3 lez vidět schéma aplikace. Editor návrhu usnadňuje propojování komponent a umí automaticky připojovat signály hodin a resetu a také připojovat známé protokoly k rozbočovačům.



Obrázek 4.3: Propojené komponenty aplikace v prostředí Xilinx Vivado.

Sobelův filtr

Sobelův filtr pro detekci hran je implementován s detekčním oknem 3x3 pixelů. Maska je pro detekci hran v ose X a Y. Obě masky jsou poté zprůměrovány. Průměrování je provedeno pomocí součtu obou masek a podělení číslem 2. Masky filtrů a výpočet je popsán zde [3]

$$\mathbf{G_x} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * I \qquad \mathbf{G_y} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * I$$

[2] http://man7.org/linux/man-pages/man2/ioctl.2.html

[3] http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html

$$\mathbf{G} = \frac{|G_x| + |G_y|}{2}$$

Komponenta je zřetězená, kde iniciační interval je 1 takt. Pracuje s pixely o bitové šířce 18 bitů. Filtr také ošetřuje okrajové hodnoty. Pokud maska je za okrajem, tak se použijí nejbližší platné hodnoty. Vstupem komponenty je AXI-Stream ze vstupními pixely, výstupem je rovněž AXI-Stream. Komponenta vždy čeká na vstupní data a poté zahájí výpočet a posunutí fronty. Přicházející pixely postupně posouvají předešlé pixely ve frontě. K uložení řádků je použita paměť BRAM. Následující tabulka 4.1 ukazuje množství spotřebovaných zdrojů v závislosti na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	2	17.36	374	1204	4	0
100	4	8.61	457	1224	4	0
150	5	5.79	569	1228	4	0
200	9	4.16	767	1228	4	0
250	12	3.58	988	1265	4	0
280	16	3.53	1166	1379	4	0

Tabulka 4.1: Závislost množství zdrojů na frekvenci u Sobelova filtru

Mediánový filtr

Mediánový filtr se používá k odstranění šumu. Nová hodnota pixelu je vypočítaný medián z okolí bodu. Proto se potlačí šum typu "pepř a sůl"(černé a bílé pixely). Okolí bodu je v této implementaci 5x5 pixelů. Celá komponenta je zřetězena s intervalem 1 takt. Při hledání mediánu je potřeba si v prvním taktu uložit ze sdílené paměti všechny body do lokálního pole a poté začít výpočet. Výpočet zabere několik hodinových taktů a to už sdílená paměť obsahuje jiná data. Porovnávání probíhá vždy mezi 2 sousedy a výsledky buď zůstanou do dalšího taktu nebo se prohodí. Každý lichý cyklus se porovnává od 0. prvku a každý sudý cyklus se porovnává od 1. prvku dále. Při okolí 5x5 pixelů je medián vypočítaný ve 24 cyklech. Tabulka 4.2 zobrazuje využití zdrojů v závislosti na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	6	17.12	2160	12179	8	0
100	12	8.74	4172	12179	8	0
150	23	5.05	7906	12179	8	0
200	24	3.68	8010	12179	8	0
250	47	3.53	15453	12396	8	0
280	47	3.53	15453	12396	8	0

Tabulka 4.2: Závislost množství zdrojů na frekvenci u mediánového filtru

Bilaterální filtr

Bilaterální filtr patří ke složitějším filtrům obrazu. Jeho cílem je vyhladit obraz, ale ponechat ostré přechody. Tato komponenta je opět zřetězena s intervalem 1 hodinový takt. Maska bilaterálního filtru je použita 11x11 pixelů. Při aplikaci masky filtru se počítá, zda daný

pixel je součástí ostrého přechodu nebo je přechod malý. Poté je provedeno prahování do několika skupin a podle výsledku je pixel buď ponechán nebo vyhlazen podle okolí.

Při implementaci na CPU jsou použity float proměnné a operace dělení a umocňování. Pro implementaci v FPGA byly použity proměnné s pevnou řádovou čárkou, mocniny byly převedeny na bitové posuvy a pro dělení byla použita tabulka s předpočítanými hodnotami, se kterými se pouze násobí. Tyto optimalizace vedly ke snížení počtu zdrojů a zkrácení latence. Optimalizace jsou popsány v kapitole 5. Následující tabulka 4.3 ukazuje využití zdrojů v závislosti na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	14	17.45	5615	19544	20	2
100	33	8.75	6726	19712	20	2
150	46	5.78	8185	20318	20	2
200	70	4.23	11339	20733	20	2
250	105	3.53	12616	20989	20	2
280	121	3.53	14131	21616	20	2

Tabulka 4.3: Závislost množství zdrojů na frekvenci u bilaterálního filtru

Změna rozlišení obrazu - Image Scale

Komponenta `ImageScale` slouží u architektury detektoru registračních značek ke změně rozlišení. Jelikož klasifikátor pracuje s určitým oknem (v tomto případě 60 x 15 px), je potřeba postupně měnit rozlišení obrazu, pokud chceme detekovat objekty různých velikostí. Komponenta zmenšuje obraz na 0.8 původního rozlišení.

Pro změnu rozlišení je použita metoda Nearest neighbor. Tato metoda pracuje tak, že vynechává v daném poměru pixely. V případě změny rozlišení na 0.8 původní velikosti se vynechá každý pátý řádek a sloupec.

Komponenta je zřetězená na vstupu a na výstupu používá rozhraní signálů. Komponenta podle signálu SOF (začátek snímku) detekuje počátek souřadnic bodů a dále počítá řádky a sloupce. Na vstupu komponenty je také parametr šířky vstupního obrazu. Jelikož může komponenta pracovat s různými vstupními velikostmi, není ji nutné znovu syntetizovat. Stačí pouze zadat parametr, který je připojený přes protokol AXI-Lite. V následující tabulce 4.4 lze vidět využití zdrojů v závislosti na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	0	9.17	86	196	0	0
100	1	6.18	100	196	0	0
150	1	5.26	105	199	0	0
200	3	4.56	128	207	0	0
210	3	4.56	128	207	0	0

Tabulka 4.4: Závislost množství zdrojů na frekvenci u komponenty pro změnu rozlišení

Zpracování výstupu z klasifikátoru - ClasifOut

Výstup z klasifikátoru je pouze 16 bitové číslo reprezentující pravděpodobnost, že se v daném bodě nachází registrační značka. Nejprve je nutné přiřadit souřadnice ke vstupní

hodnotě. Počátek souřadnic je synchronizovaný pomocí signálu SOF (začátek snímku). Dále probíhá počítání řádků a sloupců. Vstup klasifikátoru může být již několikrát zmenšen, proto je nutné přepočítat souřadnice do původního rozlišení. Výpočet se provádí násobením konstantou v pevné řádové čárce.

Dalším úkolem této komponenty je provést prahování vstupní hodnoty. Pouze hodnoty, které přesáhnou práh, jsou považovány za možný střed registrační značky. Pokud v daném snímku více hodnot přesáhne práh, pak je vybrána nejvyšší hodnota.

Komponenta je napojena na rozhraní AXI-Lite a přes tento protokol probíhá konfigurace registrů komponenty. Jedná se o nastavení prahu, nastavení konstanty pro násobení souřadnic a nastavení rozlišení vstupujících hodnot z klasifikátoru.

Výsledek komponenty je posílán jednou za snímek. Ve výsledku se posílají souřadnice bodu s největší hodnotou a dále index určující, ve kterém rozlišení se bod našel. Tato informace se posílá do jednotky vykreslující rámeček. Tabulka 4.5 ukazuje porovnání počtu zdrojů na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	1	14.2	300	573	0	2
100	2	7.64	400	574	0	2
150	4	6.60	411	597	0	2

Tabulka 4.5: Závislost množství zdrojů na frekvenci u komponenty zpracování výstupu z klasifikátoru.

Vykreslování rámečku do obrázku

K architektuře detektoru registračních značek patří také komponenta, která vykresluje rámeček kolem registrační značky do obrazu. Byla zvolena optimalizace, kdy se vykresluje rámeček až do následujícího snímku videa. Vykreslování do právě zpracovávaného snímku by bylo paměťově náročné a paměť BRAM by byla nedostačující.

Komponenta přijímá video přes rozhraní AXI-Stream a zároveň přijímá informace o středu rámečku a o indexu velikosti. V komponentě je vestavěná logika, která podle indexu spočítá velikost rámečku. Komponenta také počítá souřadnice řádků a sloupců a na odpovídající místa posílá bílou barvu místo hodnoty pixelu. V tabulce 4.6 je zobrazena závislost množství zdrojů na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	1	16.59	149	459	0	0
100	2	8.37	179	461	0	0
150	3	5.63	167	481	0	0
200	4	4.26	171	482	0	0
250	7	3.53	333	485	0	0
280	7	3.53	333	485	0	0

Tabulka 4.6: Závislost množství zdrojů na frekvenci u jednotky vykreslující rámeček do videa.

Komponenta pro hledání začátku paketu

Komponenta je implementována jako nekonečná zřetězená smyčka, která se může nacházet ve dvou stavech. První stav je, kdy přicházejí byty ze vstupu, na konci linky se byty posílají na výstup a počítá se kontrolní součet a hledá se začátek paketu. Druhý stav je v případě, že se již našel začátek a je potřeba odeslat hlavičku. V tomto případě je pozastaveno čtení ze vstupu a linka se neposouvá.

Komponenta je ukázkou netriviálního zpracování dat. Obrazové filtry pracovaly vždy tak, že jeden pixel přijaly a jeden pixel byl na výstupu. Tato komponenta má vnitřní paměť a může být ve stavu, kdy jen posílá, nebo jen přijímá.

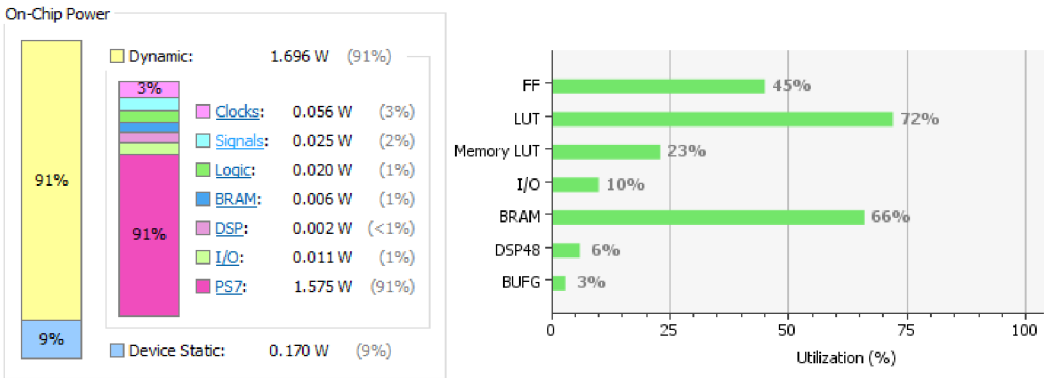
Rozhraním komponenty je AXI-Stream na vstupu a na výstupu. AXI-Stream má 8-bitová data a 1 bitový signál LAST, který označuje konec paketu. Pozastavení linky je provedeno tak, že na vstupu není aktivní signál READY. Platná data na výstupu mají aktivní signál VALID. V následující tabulce 4.7 je zobrazena závislost počtu zdrojů na frekvenci.

f [MHz]	latence	perioda [ns]	FF	LUT	BRAM 18K	DSP48E
50	2	16.68	721	668	0	0
100	3	7.69	796	679	0	0
150	4	5.61	847	683	0	0
204	5	4.88	932	700	0	0

Tabulka 4.7: Závislost množství zdrojů na frekvenci u komponenty pro nalezení začátku paketu.

Využití zdrojů a příkon celé aplikace

V prostředí Vivado po sestavení celého návrhu aplikace z komponent lze spustit logickou syntézu a implementaci na konkrétní čip. Na vývojové desce byl použit Xilinx ZYNQ Z-7020. Po implementaci lze analyzovat a kontrolovat výsledky. Na obrázku 4.4 lze vidět grafické znázornění příkonu a využití zdrojů čipu. FPGA v aplikaci bylo taktováno na 75 MHz a ARM procesor na 666 MHz. Z grafu je patrné, že asi 91% příkonu odebírá ARM procesor.



Obrázek 4.4: Celkový příkon a využití zdrojů aplikace.

4.2 Porovnání implementace v C/C++ a HDL jazycích

Programování FPGA pomocí jazyka C/C++ a nebo pomocí HDL jazyků přináší značné rozdíly v přístupu k řešení problému. Srovnáním se zabývá také několik vědeckých článků, jejichž výsledky byly uvedeny v kapitole 2.1. Rovněž v této práci jsem mohl porovnat implementaci bilaterálního filtru a komponenty pro hledání začátku paketu v jazyce C++ a VHDL. Počty zdrojů jsou uvedeny po logické syntéze v prostředí Xilinx Vivado.

Srovnání bilaterálního filtru

V této práci byl implementován bilaterální filtr pomocí jazyka C++ a prostředí Xilinx Vivado HLS. Bilaterální filtr rovněž implementoval Ing. Martin Musil pomocí jazyka VHDL. V následující tabulce 4.8 jsou vidět výsledky v počtu zdrojů u obou řešení. V obou případech byla provedena logická syntéza pomocí prostředí Xilinx Vivado.

	FF	LUT	Mem LUT	BRAM	DSP
VHDL	7810	7480	111	9	1
HLS	10617	9817	1158	10	2

Tabulka 4.8: Porovnání Bilaterálního filtru - VHDL implementace a HLS implementace.

Přesto, že se implementace mírně lišily (například v HLS bylo použito přesnějšího dělení s tabulkou), tak podle tabulky 4.8 je vidět, že VHDL implementace je efektivnější, co se týče spotřeby zdrojů. Odhad času vývoje zkušeného vývojáře ve VHDL, je asi 14 dní práce. Jelikož jsem s prostředím Vivado HLS začínal, tak implementace mi zabrala více času. Odhaduji však, že kdybych nyní řešil podobný problém se současnými znalostmi, tak implementace může být do týdne hotová.

Srovnání komponenty pro hledání začátku paketu

Další komponenta, kterou jsem měl možnost porovnat s řešením Ing. Martina Musila, byla komponenta pro hledání začátku paketu. Tato komponenta má vnitřní vyrovnávací paměť 40 bytů.

	FF	LUT	Mem LUT	BRAM	DSP	max f
VHDL	844	606	140	0.5	0	333 MHz
HLS	991	459	4	0	0	204 MHz

Tabulka 4.9: Porovnání komponenty pro hledání začátku paketu - VHDL implementace a HLS implementace.

VHDL implementace obsahuje navíc ještě 2kB frontu (zabírá 0.5 BRAM) pro uchování celého paketu. VHDL implementace dosáhla vyšší frekvence, zdroje potřebné pro obě implementace jsou srovnatelné. Odhad časové náročnosti u VHDL implementace je asi 13 hodin, u HLS implementace asi 40 hodin. Nejtěžší na celé implementaci bylo vyřešit posouvání fronty, když nepřicházejí pakety, což zabralo asi 80% času. Řešením bylo použití nekonečné zřetězené smyčky a neblokujícího čtení, což je popsáno v kapitole 5.1. Toto řešení přineslo další zkušenosti pro vývoj a lze jej využít i pro další úlohy zpracování dat.

Zhodnocení přístupů

V této podkapitole bylo ukázáno na reálných případech, jaké jsou výsledky spotřebovaných zdrojů při implementaci při použití HLS a HDL přístupu. Z výsledků je patrné, že HLS přístup zabírá více zdrojů a dosahuje menší výkonnosti. Oproti tomu přináší úsporu času při opakovaných implementacích podobných problémů. Také zdrojový kód je přenositelný do dalších aplikací. Nevýhodou pro vývojáře také může být licence dalšího vývojového prostředí. Jelikož Xilinx Vivado je relativně nový produkt (od roku 2012), tak vývojové prostředí se vyvíjí a přidává se funkcionalita. Rovněž se rozšiřuje podpora pro vývojáře v podobě dokumentací i komunitních příspěvků v rámci internetového fóra.

V následující tabulce 4.10 jsou shrnuty výhody a nevýhody při použití syntézy na systémové úrovni.

Výhody HLS	**Nevýhody HLS**
• znovupoužití bloků v C/C++ • rychlejší návrh při řešení opakovaných problémů • rychlá simulace kódu C na běžném CPU • snadnější aplikace změn • rychlý průzkum mikroarchitektur • práce s rozhraním AXI • práce s poli a smyčkami	• nemožnost plně řídit plán operací • v C/C++ nelze přistupovat k signálům CLK a RESET • více spotřebovaných zdrojů • menší maximální frekvence • licence vývojového prostředí • slabší dokumentace a příklady

Tabulka 4.10: Porovnání HDL a HLS implementace.

4.3 Testování aplikace

Při vývoji aplikací na FPGA patří testování a simulace k náročným úkolům. Při vývoji softwaru pro běžné CPU lze program přeložit a spustit většinou v krátkém čase a vyhodnotit výsledky. Při programování FPGA je situace složitější, protože syntéza a implementace obvodu jsou časově náročné operace. Testovat aplikaci lze na úrovni jazyka C/C++, na úrovni RTL, nebo také ladit spuštěnou aplikaci na vývojové desce.

Využil jsem několik úrovní a způsobů simulace a testování obvodu. V následujících podkapitolách budou popsány jednotlivé možnosti a jejich výhody a nevýhody. Při vývoji komponent jsem použil nejprve simulaci kódu v C/C++. V případě, kdy nebylo zřejmé chování komponenty na úrovni signálů, jsem použil behaviorální simulaci a analyzoval průběh signálů. Ladící jádro je vhodné při ladění programu nebo pro reálné sledování signálů na čipu. Každá složitější komponenta v aplikaci obsahuje Test-Bench soubor pro simulaci v C/C++.

Jelikož se jedná o aplikaci na zpracování videa, výstup z aplikace lze sledovat na monitoru, zda odpovídá požadavkům. V závěru této kapitoly jsou umístěny snímky z výstupního monitoru.

Simulace kódu v C/C++

Tuto možnost simulace přináší vývojové prostředí Xilinx Vivado HLS. Vývojář zde může napsat Test-Bench aplikaci v jazyce C/C++, ve které se volá top funkce komponenty. Lze tak provést verifikaci algoritmu, který chceme syntetizovat. Je to nejrychlejší způsob simulace, protože se neprovádí syntéza, ale pouze se spustí algoritmus na procesoru.

Tento způsob simulace je obvyklý pro syntézu na systémové úrovni. Zpravidla máme referenční algoritmus pro CPU (tzv. Golden design) a provádíme srovnání s upraveným algoritmem pro FPGA. Výstupy z algoritmů mohou být vypsány do terminálu nebo do souboru. V aplikaci jsem také využíval knihovny OpenCV ke zobrazení obrázků.

Při simulaci lze použít i nesyntetizovatelné konstrukce (např. funkce z STL jako `printf()` a podobně), takže je vhodná tato technika i pro ladění kódu. Nevýhodou tohoto přístupu je, že vývojář nemá žádné informace o tom, zda daný algoritmus lze syntetizovat, kolik taktů bude zabírat a podobně. FIFO fronty jsou v simulacích nekonečné, rovněž signály READY a VALID nejsou brány v úvahu. Jde čistě o kontrolu algoritmu, zda dává správné výsledky. Zdrojový kód 4.2 ukazuje příklad využití Test-Bench souboru k simulaci aplikace pro zpracování obrazu.

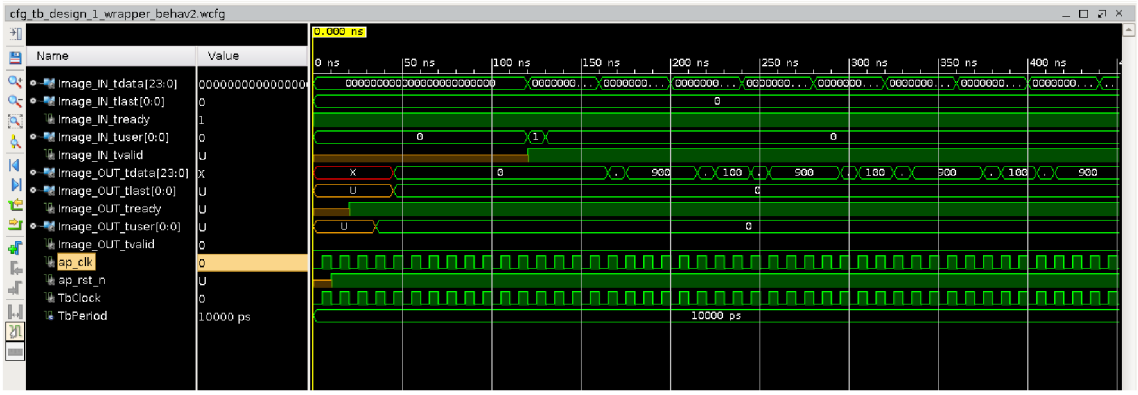
```
1  int main(){
2    Mat img_src = imread(IMAGE_IN , CV_LOAD_IMAGE_COLOR);
3    // volání referenčního algoritmu pro CPU
4    referenceAlgorithm(img_src, img_dst_1);
5    // volání algoritmu uprovaného pro HW
6    hardwareAlgorithm(img_src, img_dst_2);
7    // výsledky lze vizuálně porovnat pomocí obrázků
8    imwrite( IMAGE_OUT_REF, img_dst_1);
9    imwrite( IMAGE_OUT_HW, img_dst_2);
10 }
11
12 void hardwareAlgorithm(Mat img_src, Mat img_dst_2){
13   // simulovani AXI-Stream
14   MyAxiStream Image_IN,Image_OUT;
15
16   for (int i = 0; i < FRAME_HEIGHT ; i++) {
```

```
17      for (int j = 0; j < FRAME_WIDTH; j++) {
18        //start of frame = user
19        if(j==0 && i==0){
20          Image_IN.user = 1;
21        }
22        // end of line
23        else if(j == FRAME_WIDTH-1){
24          Image_IN.last = 1;
25        }
26        else{
27          Image_IN.user = 0;
28          Image_IN.last = 0;
29        }
30        Image_IN.data = img_src.at<uint32_t>(i,j);
31        // volání syntetizovatelné funkce s rozhraním AXI-Stream
32        topFunction(&Image_IN, &Image_OUT);
33        img_dst_2.at<uint32_t>(index) = Image_OUT.data.to_int();
34      }
35    }
36  }
```

Zdrojový kód 4.2: Ukázka Test-Bench aplikace.

Behaviorální simulace

Další možnost verifikace komponent je Behaviorální simulace v prostředí Xilinx Vivado. Tento způsob jsem používal po otestování komponenty pomocí Test-Bench aplikace v C/C++. Hlavním cílem testování je analyzovat průběh signálů v čase na vstupu a výstupu komponenty. Behaviorální simulace je časově náročnější, protože po každé změně v chování komponenty je potřeba provést syntézu na systémové úrovni a poté vygenerovat RTL popis komponenty v prostředí Vivado HLS. Až poté v prostředí Vivado aktualizovat komponentu v návrhu a spustit simulaci. Pro simulaci se také píše Test-bench aplikace ve VHDL, která nastavuje hodnoty signálů na vstupu komponenty.



Obrázek 4.5: Ukázka průběhů signálů při behaviorální simulaci.

Ladící jádro v HW (Debug Core)

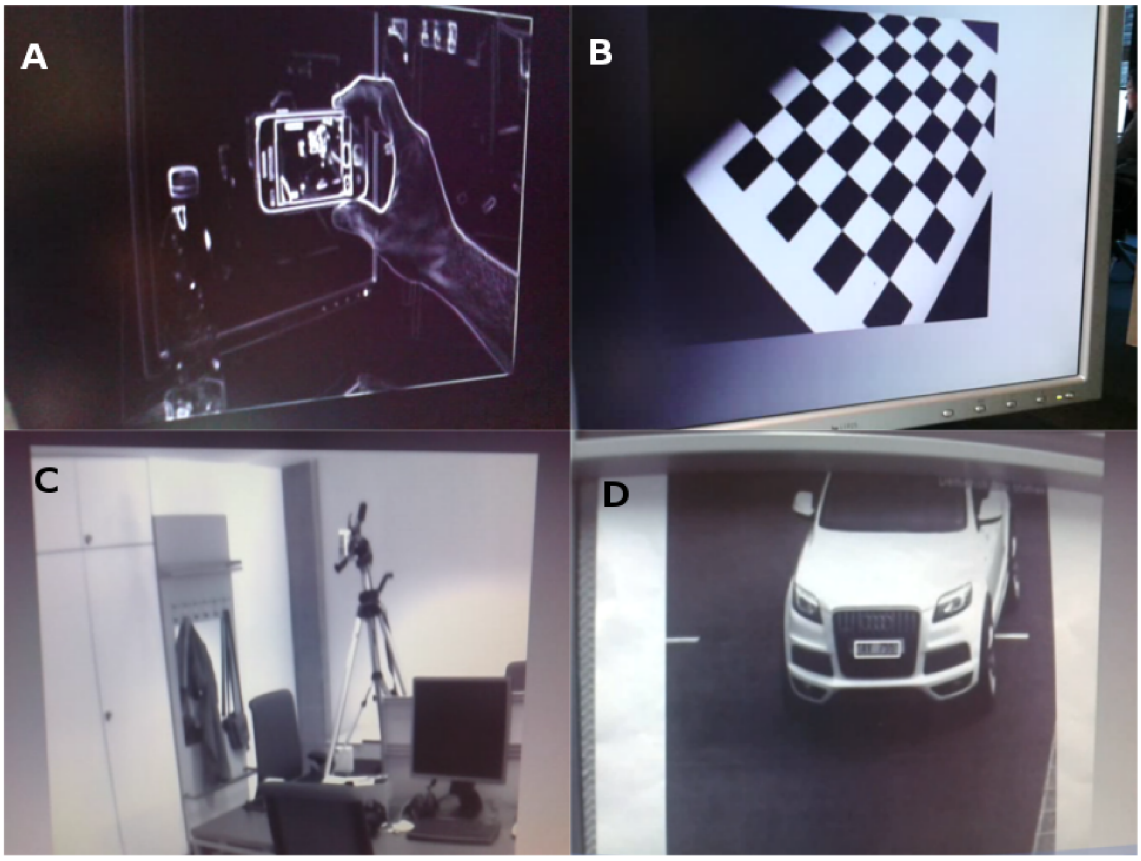
Další možností jak ladit nebo testovat aplikaci je použití ladícího jádra. Tento způsob je možné provést s vývojovým prostředím Vivado. Nejprve je nutné provést logickou syntézu

celé aplikace. Poté je možno označit signály v návrhu, které budou sledovány a vytvoří se ladící jádro jako další komponenta v FPGA, do kterého jsou přivedeny tyto signály. Jádro má omezenou velikost hodnot, které si uchová (v řádu tisíců vzorků na signál). Poté je potřeba provést implementace obvodu a naprogramování čipu. Přes rozhraní JTAG lze nastavovat triggery nebo ručně spouštět záznam průběhů signálu do ladícího jádra. Poté jsou hodnoty přeposlány do PC, kde lze v grafu analyzovat průběhy.

Tento způsob testování je časově náročný, protože je nutné provést syntézu a implementaci celé aplikace. Nicméně umožňuje sledovat signály přímo za běhu aplikace, což může být užitečné pro ladění i testování aplikace. Jedná se o výborný nástroj, který jsem několikrát použil při vývoji aplikace, kdy aplikace nefungovala a bylo náročné odhalit, kde se nachází problém.

Ukázka výstupu z aplikace

Následuje několik obrázků, které zachycují výstup na monitoru po aplikaci filtrů nebo detektoru SPZ. Rovněž je možné shlédnout demonstrační video [4].



Obrázek 4.6: A: Sobelův filtr, B: Mediánový filtr na šachovnici, která obsahuje šum pepř a sůl, C: Bilaterální filtr, D: Detekce SPZ.

[4] https://youtu.be/bm-d2kwY8AQ

Kapitola 5

Optimalizace aplikace

V této kapitole budou popsány techniky a optimalizace, které byly použity při implementaci komponent v prostředí Xilinx Vivado HLS. Také bude ukázán vliv jednotlivých optimalizací na spotřebu zdrojů a na propustnost komponenty.

5.1 Optimalizace propustnosti

Zřetězení funkcí

Častou optimalizací komponent je zřetězení operací (pipelining). Zřetězená funkce má velkou propustnost. Latence ale může být několik taktů a funkce může mít několik hodnot rozpracovaných. Při návrhu v HDL jazycích je potřeba ručně naplánovat operace do jednotlivých taktů. Což sice přináší kontrolu nad plánem operací, nicméně je to časově náročné při návrhu a nebo při aplikaci změn.

V prostředí Vivado HLS lze provést zřetězení pomocí jedné direktivy. Aplikace automaticky hledá plán operací, aby byl splněn požadovaný interval.

```
1 void sobel(AxiStream *Image_IN, AxiStream *Image_OUT){
2 #pragma HLS PIPELINE II=1
3   ...
4 }
```

Zdrojový kód 5.1: Direktiva pro zřetězení top-funkce s inicializačním intervalem 1.

V praxi však není triviální navrhnout kód v C/C++, který lze zřetězit. Překážkou pro vytvoření takového kódu jsou zejména datové závislosti statických proměnných mezi jednotlivými iteracemi. Při návrhu a implementaci je algoritmus většinou rychle popsán v jazycích C/C++, ale při syntéze často nastává problém, kdy nelze zřetězit funkci se zadaným intervalem a nástroj Vivado HLS postupně hledá nejbližší plán k požadované propustnosti. V komponentách pro zpracování videa je požadovaná co nejvyšší propustnost a proto jsou komponenty optimalizovány na inicializační interval 1 hodinový takt. V následujících příkladech budou uvedeny problémy, které bránily zřetězení komponent a také jakým způsobem byly vyřešeny.

Falešná závislost u polí

V obrazových filtrech je použita paměť BRAM jako FIFO pro uložení řádků kolem aktuálně zpracovávané oblasti. Paměť BRAM je dvouportová a každý takt lze číst nebo zapisovat dvě hodnoty současně. Při pokusu o zřetězení se však objevil problém při dosažení inicializačního

intervalu jeden takt. Na obrázku 5.1 lez vidět výpis při plánování obvodu v prostředí Vivado HLS.

```
@I [HLS-10] ---------------------------------------------------------------
@I [HLS-10] -- Scheduling module 'sobel'
@I [HLS-10] ---------------------------------------------------------------
@I [SCHED-11] Starting scheduling ...
@I [SCHED-61] Pipelining function 'sobel'.
@W [SCHED-68] Unable to enforce a carried dependency constraint (II = 1, distance = 1)
   between 'store' operation (SobelFilter/sobel.h:52->SobelFilter/sobel.cpp:42) of variable
'window_V_2_load' on array 'buffer0_data_V' and 'load' operation ('buffer0_data_V_load', SobelFilter/
sobel.h:51->SobelFilter/sobel.cpp:42) on array 'buffer0_data_V'.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 6.
@I [SCHED-11] Finished scheduling.
```

Summary

Latency		Interval		
min	max	min	max	Type
5	5	2	2	function

Obrázek 5.1: Problém při aplikaci optimalizace zřetězení. Nelze dosáhnout požadované propustnosti u BRAM.

Řešením tohoto problému je přidání direktivy pro zrušení závislosti (direktiva dependence), která ručně nastavuje závislosti v rámci polí. V rámci iterace se přistupuje ke dvěma prvkům pole v jednom taktu. Z jednoho se čte a do druhého zapisuje. Do pole se přistupuje pomocí rotujících indexů. Proto nástroj nedovolil zřetězit na interval 1, protože z popisu nebylo zaručeno, že se nebude zapisovat a číst ze stejného prvku. Po ručním nastavení nezávislosti mezi prvky pole již nástroj provede plánování obvodu s požadovanou propustností 1 takt.

```
1 static bram_shift<BUFFER_SIZE> buffer;
2 #pragma HLS DEPENDENCE variable=buffer.data array inter false
```

Zdrojový kód 5.2: Nastavení nezávislosti v přístupu k poli.

Několik přístupů ke globálním proměnným v iteraci

Při návrhu algoritmu na CPU není zpravidla potřeba se zamýšlet nad počtem přístupů ke globálním proměnným. U zřetězené aplikace pro FPGA při popisu v C/C++ je to však jeden z hlavních problémů. Ke globálním proměnným nelze přistupovat neomezeně, jelikož další iterace zřetězení rovněž bude přistupovat k těmto datům. Pokud je vytvářena komponenta s intervalem 1 takt, tak je třeba brát v úvahu, že nelze globální proměnnou používat déle jak jeden takt. Jinak dojde k přepsání dat další iterací. Čtení a zápis do globální proměnné nemusí být nutně ve stejný hodinový takt, ale pokud to datové závislosti dovolí, tak mezi operacemi může být nějaká latence, ale musí být u všech iterací stejná. Následující příklad ukazuje počítadlo řádků a sloupců, které bylo použito u komponent filtrů obrazu i architektury klasifikátoru AdaBoost.

Ve zdrojovém kódu 5.3 můžeme vidět intuitivní implementaci algoritmu. Tato implementace však nelze zřetězit s intervalem 1, jelikož je naplánováno násobné přepsání proměnné a není dosažen požadovaný čas jednoho hodinového cyklu. Řešení, které lze vidět ve zdrojovém kódu 5.4, již lze zřetězit s intervalem 1 a frekvence obvodu je dostatečná. Algoritmus počítá řádky a sloupce na základě signálu SOF (Start Of Frame).

```
static ap_int<13> row = 0;
static ap_int<13> column = 0;

//pixel counter
column++;
if(SOF == 1){
  row = 0;
  column = 0;
}
else if(column == FRAME_WIDTH){
  column = 0;
  row++;
}
```

Zdrojový kód 5.3: Přístup ke globálním proměnným - zřetězené řešení nesplňuje požadovanou časovou periodu.

```
static ap_int<13> row = 0;
static ap_int<13> column = 0;

//pixel counter
if(SOF == 1){
  row = 0;
  column = 0;
}
else if(column == FRAME_WIDTH-1){
  column = 0;
  row++;
}
else{
  column++;
}
```

Zdrojový kód 5.4: Přístup ke globálním proměnným - správné řešení.

V některých případech je nutné provádět s globální proměnnou operace na několik hodinových taktů. Pokud to algoritmus dovoluje, je možné si globální proměnnou zkopírovat do lokální proměnné v iteraci. Poté lze přistupovat k proměnné i několik hodinových taktů.

V následující tabulce 5.1 lze vidět výsledky propustnosti u obrazových filtrů bez použití optimalizace zřetězení. Frekvence byla na 150 MHz, pro porovnání se zřetězenou implementací v kapitole 4.1. Z tabulky je zřejmé, že neřetězená implementace prodlouží několikanásobně propustnost a také se zmenší nároky na zdroje.

Filtr	FF	LUT	BRAM	DSP	**Interval**
Sobelův	653	1120	4	2	**106**
Mediánový	1385	1173	9	3	**1437**
Bilaterální	4836	2950	22	2	**757**

Tabulka 5.1: Počet zdrojů a propustnost u neřetězené implementace obrazových filtrů.

Nekonečná zřetězená smyčka

Při implementaci obrazových filtrů nebo komponent pro detektor registračních značek byly použity zřetězené komponenty, které blokujícím způsobem čekaly na platný vstup a poté vykonaly určitou akci. Například při filtrování obrazu jeden příchozí pixel posunul linku a poslední pixel se dostal na výstup. Pokud přestaly přicházet vstupní pixely, tak komponenta stála a část rozpracovaných pixelů zůstala v lince. U videa takové chování nevadí. Jiná je situace u síťového provozu. Část posledního paketu by totiž zůstávala v lince, což je ovšem nežádoucí, protože by došlo ke ztrátě dat.

Řešením situace, kdy je potřeba posouvat linku, aniž by přišel další příchozí byte je vyřešena v komponentě pomocí nekonečné zřetězené smyčky s inicializačním intervalem jeden takt. Pro čtení z rozhraní AXI4-Stream je použita neblokující varianta funkce `read`.

```
void AxiEthernetPacket(hls::stream<AxiStream> *DATA_IN, hls::stream<
    AxiStream> *DATA_OUT)
{
#pragma HLS INTERFACE axis port=DATA_OUT
#pragma HLS INTERFACE axis port=DATA_IN

  init(); // vykoná se pouze 1x po resetu

  INF_LOOP: while(1){ //pro simulace v C je třeba požít omezený for cyklus
  #pragma HLS PIPELINE II=1

    AxiStream in,out;
    ap_uint<1> readSucc = DATA_IN->read_nb(in); // neblokující čtení
    if(readSucc){
      ... // byl přečten vstup
    }
    ...
  }
}
```

Zdrojový kód 5.5: Nekonečná zřetězená smyčka u komponenty pro vyhledání začátku paketu.

5.2 Optimalizace plochy na čipu

Dělení s tabulkou

Po syntéze na systémové úrovni lze výsledný plán analyzovat. V něm vidíme, kolik taktů potřebují jednotlivé operace. Dělení je náročná operace na zdroje. Například u bilaterálního filtru bylo potřeba provést dělení dvou čísel, kdy dělitel bylo desetinné číslo v pevné řádové čárce. Samotná operace trvala 31 taktů.

```
ap_uint<25> sumShift = 0;
ap_ufixed<10,4,AP_RND_ZERO> normShift = 0;
...
ap_uint<36> result_temp = sumShift / normShift;
```

Zdrojový kód 5.6: Neoptimalizované dělení pomocí operátoru.

Možnost, jak urychlit latenci a ušetřit plochu na čipu je předpočítat si tabulku pro dělení. Výsledek je ukázán ve zdrojovém kódu 5.7. Dělitel v tomto případě je 10 bitové číslo, takže celkově se jedná o 1023 možností (bez 0). Do tabulky se uloží převrácená a posunutá hodnota všech možných dělitelů. Při dělení pak stačí načíst hodnotu z tabulky, vynásobit dělenec a hodnotu posunout. Výsledná operace trvá 5 taktů - jedná se o násobení na DSP. Tabulka předpočítaných hodnot zabírá 1 BRAM. Tento výpočet by šel ještě optimalizovat a použít i menší přesnost hodnot v tabulce.

```
static ap_uint<23> div_table[1024] = {
  1, 4194304, 2097152, 1398101, 1048576, 838860, 699050, 599186, 524288,
  466033, 419430, 381300, 349525, 322638, 299593, 279620, 262144, 246723,
  233016, 220752, 209715, 199728, 190650, 182361, 174762, 167772, 161319,
  ...
  ...
}
ap_uint<25> sumShift = 0;
ap_ufixed<10,4,AP_RND_ZERO> normShift = 0;
...
//deleni s tabulkou: ap_uint<36> result = sumShift / normShift;
ap_ufixed<16,10,AP_RND_ZERO> index = normShift; // index do tabulky
index = index << 6;
ap_uint<36> result_temp = sumShift * div_table[index]; // nasobeni na DSP

// zaokrouhleni deleni
result_temp = result_temp >> 15;
ap_uint<1> add = result_temp & 1;
result_temp = result_temp >> 1;
result_temp += add;
data_OUT = result_temp;
```

Zdrojový kód 5.7: Optimalizované dělení pomocí tabulky.

Následující tabulka 5.2 zobrazuje srovnání při použití optimalizace dělení s tabulkou a nebo použití operátoru dělení u bilaterálního filtru při syntéze na 150 MHz.

	FF	LUT	BRAM	DSP	Latence
Operátor '/'	9549	21165	18	0	73
Tabulka	8253	20318	20	2	46

Tabulka 5.2: Srovnání počtu zdrojů při použití tabulky a operátoru dělení.

Desetinná čísla

Při psaní algoritmů ve vestavěných zařízeních se často používá převod čísel na celočíselné hodnoty, aby se nemusela používat desetinná čísla. Rovněž v FPGA je zpravidla nejefektivnějším řešením použít celá čísla. Nicméně v praxi je také potřeba pracovat i s desetinnými čísly. Prostředí Vivado HLS podporuje čísla s plovoucí řádovou čárkou (`float`, `double`) i s pevnou řádovou čárkou (`ap_fixed`). Na následujícím příkladu bude ukázána optimalizace plochy na čipu při použití nejprve proměnných s plovoucí řádovou čárkou a poté s pevnou řádovou čárkou u bilaterálního filtru.

	FF	LUT	BRAM	DSP	Latence	Interval
Float point	22650	352616	18	2242	46	1
Fixed point	8185	20318	20	2	771	1

Tabulka 5.3: Srovnání počtu zdrojů při použití float nebo fixed point aritmetiky u bilaterálního filtru.

Z tabulky 5.3 je patrné, že při použití plovoucí řádové čárky u zřetězené implementace bilaterálního filtru by vedlo k vysoké spotřebě zdrojů. Počet zdrojů by v takovém případě několikanásobně překročil počet dostupných prostředků na čipu a taková implementace by v praxi nebyla použitelná. Na FPGA má smysl použití plovoucí řádové čárky jen v nezbytných případech a pouze u jednoduchých výpočtů.

Optimalizace algoritmu

V některých případech lze také optimalizovat samotný algoritmus. Například u Sobelova filtru se počítá konvoluce ve dvou směrech. V ose X a Y. Výsledný gradient však je potřeba vypočítat z obou hodnot. Jednou z možností je vypočítání odmocniny součtu druhých mocnin gradientů v ose X a Y. Další z možností, která se používá u Sobelova filtru je méně náročná metoda, kdy se zprůměruje součet absolutních hodnot gradientu. V tabulce 5.4 lze vidět srovnání obou těchto přístupů.

	FF	LUT	BRAM	DSP	Latence	Interval
Odmocnina	888	2596	4	4	19	1
Průměr abs. hodnot	612	1233	4	0	6	1

Tabulka 5.4: Srovnání počtu zdrojů při použití odmocniny nebo průměru absolutních hodnot.

Kapitola 6

Závěr

Cílem diplomové práce bylo pomocí syntézy na systémové úrovni navrhnout a implementovat aplikaci pro zpracování obrazu. Všechny body stanovené v zadání se podařilo splnit. První bod zadání byl prostudovat problematiku syntézy na systémové úrovni, platformu Xilinx ZYNQ a vývojové prostředí Xilinx Vivado. Tento teoretický úvod je popsán v kapitole 2. V kapitole jsou uvedeny principy syntézy na systémové úrovni a její jednotlivé fáze. Také je zde popsána platforma Xilinx ZYNQ, její části a rozhraní. Poslední část teoretického úvodu se zabývá vývojovým prostředím Vivado HLS. Vývojář zde nalezne souhrn technik, které lze použít ve vývojovém prostředí. Text je doplněn pro přehlednost ukázkami zdrojových kódů.

Druhým bodem zadání bylo navrhnout aplikaci v oblasti zpracování obrazu. Popis navržené aplikace je v kapitole 3. Aplikace je rozdělena mezi ARM procesor a FPGA. Vstupem obrazu je kamera nebo obrázek. Dále uživatel může zvolit mezi několika možnostmi zpracování obrazu. V aplikaci je implementován Sobelův filtr pro detekci hran, mediánový filtr pro odstranění šumu, bilaterální filtr pro vyhlazení obrazu a také architektura ke klasifikátoru AdaBoost pro detekci registračních značek vozidel. Jako rozšíření diplomové práce je také navržena a implementována komponenta pro hledání začátku paketu v síťovém provozu.

Třetí úkol práce bylo implementovat a otestovat aplikaci na platformě Xilinx ZYNQ. Výsledky implementace a způsob testování je popsán v kapitole 4. Ke každé složitější komponentě je uvedena tabulka s potřebnými zdroji pro několik frekvencí, až po maximální dosaženou frekvenci. Kapitola také obsahuje srovnání implementace pomocí syntézy na systémové úrovni a implementace pomocí HDL jazyků.

Čtvrtý úkol byl navrhnout a provést optimalizace aplikace. Použité optimalizace jsou popsány v kapitole 5. Optimalizace byly provedeny buď s cílem vyšší propustnosti komponenty nebo kvůli ušetření plochy na čipu. K optimalizacím je uvedena tabulka pro srovnání s výsledky implementace bez provedení optimalizací.

S částí této práce jsem se také účastnil konference Excel@FIT 2015 a práce byla přijata do sborníku konference.

Práce zkoumá využití syntézy na systémové úrovni pomocí jazyka C/C++ a vývojového prostředí Xilinx Vivado na platformě ZYNQ. Nabízí se zde hned několik směrů, jakým způsobem by šlo na práci navázat. Zajímavým pokračováním práce by bylo prostudovat i další způsoby syntézy na systémové úrovni. Například jazyk SystemC nebo framework OpenCL a také se seznámit s dalšími vývojovými nástroji. Dalším směrem může být výzkum v oblasti akcelerace algoritmů na platformě Xilinx ZYNQ. Například možnost využití jednotky NEON pro akceleraci multimédií na procesoru ARM nebo využití FPGA. Také další možností je provést výzkum v oblasti výpočtů s nízkým příkonem na této platformě.

Literatura

[1] Crockett, L.; Elliot, R.; Enderwitz, M.: *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014, ISBN 9780992978709.
URL http://www.zynqbook.com/

[2] Elliott, J. P.: *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1999, ISBN 079238542X.

[3] Filip, K.: *Implementace obrazových klasifikátorů v FPGA*. Diplomová práce, VUT Brno FIT, 2010.

[4] Martin, M.: *Přenosy rastrových dat v FPGA*. Diplomová práce, VUT Brno FIT, 2012.

[5] McFarland, M. C.; Parker, A. C.; Camposano, R.: Tutorial on High-level Synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, ISBN 0-8186-8864-5, s. 330–336.
URL http://dl.acm.org/citation.cfm?id=285730.285784

[6] Tomáš, M.: Pokročilé metody syntézy číslicových obvodů (High Level Synthesis). VUT Brno FIT, Přednáška kurzu PCS, 2014.

[7] Winterstein, F.; Bayliss, S.; Constantinides, G.: High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, s. 362–365.

[8] Xilinx: AXI4-Stream Video IP and System Design Guide. 2014.
URL http://www.xilinx.com/support/documentation/ip_documentation/axi_videoip/v1_0/ug934_axi_videoIP.pdf

[9] Xilinx: Vivado Design Suite User Guide High-Level Synthesis. 2014.
URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug902-vivado-high-level-synthesis.pdf

[10] Xilinx: Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit. 2014.
URL http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

[11] Xilinx: ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide. 2014.
URL http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

[12] Xilinx: LogiCORE IP AXI Interconnect Product Guide. 2015.

[13] Zwagerman, M. D.: *High Level Synthesis, a Use Case Comparison with Hardware Description Language*. Diplomová práce, Grand Valley State University, 2015.

Seznam zkratek

Zkratka	Význam	Zkratka	Význam
ADC	Analog to Digital Converter	LED	Light-Emitting Diode
ALAP	As Late As Possible	LUT	Lookup Table
APU	Application Processing Unit	MIO	Multiplexed Input / Output
ARM	architektura procesorů	MUX	Multiplexor
ASAP	As Soon As Possible	OCM	On Chip Memory
AXI	Advanced eXtensible Interface	OpenCV	Open source Computer Vision
BRAM	Block RAM	OTG	On The Go
CAN	Controller Area Network	PC	Personal Computer
CDFG	Control / Data Flow Graph	PL	Programmable Logic
CLB	Configurable Logic Block	PS	Processing System
CPU	Central Processing Unit	RAM	Random Access Memory
DDR	Double Data Rate	RAW	Read After Write
DEMUX	Demultiplexor	RGB	Red - Green - Blue
DSP	Digital Signal Processor	ROM	Read Only Memory
EMIO	Extended Multiplexed Input / Output	RTL	Register Transfer Level
EOL	End Of Line	SATA	Serial Advanced Technology Attachment
ESL	Electronic System-Level	SD	Secure Digital
FF	Flip Flop	SDK	Software Developer's Kit
FIFO	First In First Out	SoC	System on Chip
FMC	FPGA Mezzanine Card	SOF	Start Of Frame
FPGA	Field Programmable Gate Array	SPI	Serial Peripheral Interface
FPU	Floating Point Unit	SPZ	státní poznávací značka
FSM	Finite State Machine	SRL	Shift Register LUT
GPIO	General Purpose Input / Output	STL	Standard Template Library
GPS	Global Positioning System	TCL	Tool Command Language
HD	High Definition	TLM	Transaction Level Model
HDL	Hardware Description Language	UART	Universal Asynchronous Receiver Transmitter
HDMI	High Definition Multimedia Interface	ULP	Unit in the Last Place
HLS	High Level Synthesis	USB	Universal Serial Bus
I2C	Inter-Integrated Circuit	VHDL	VHSIC Hardware Description Language
II	Initiation Interval	WAR	Write After Read
IP core	Intellectual Property core	WAW	Write After Write
JTAG	Joint Test Action Group		

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojové kódy aplikace. Aplikace je uložena ve třech složkách. První složka obsahuje zdrojové kódy pro procesor ARM. Další složba obsahuje jednotlivé komponenty, které byly vytvořeny v prostředí Xilinx Vivado HLS verze 2014.4. Poslední složka obsahuje celkový návrh aplikace v prostředí Xilinx Vivado.

Adresář	**Popis**
`ARM`	Obsahuje zdrojové kódy aplikace pro ARM procesor
`Komponenty HLS`	Zdrojové kódy komponent pro FPGA, vytvořeno v prostředí Vivado HLS
`Video Ukazka`	Video s ukázkou chování aplikace
`Vivado Design`	Zdrojové kódy celé aplikace v prostředí Vivado