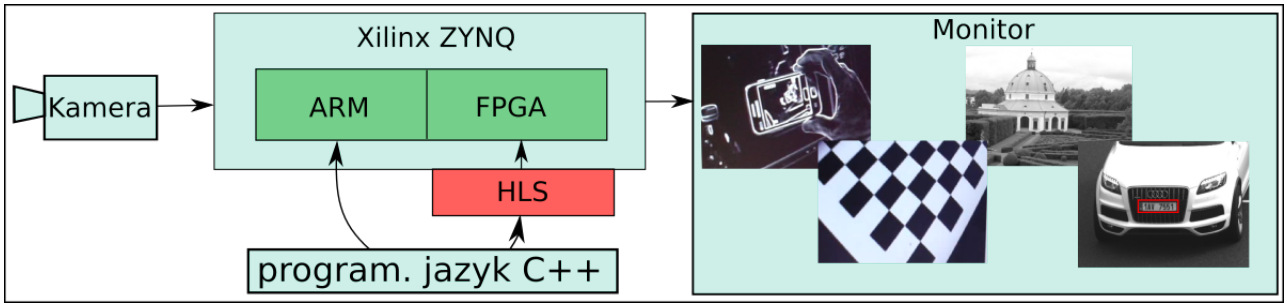
Excel@FIT 2015
http://excel.fit.vutbr.cz

Aplikace s využitím syntézy na systémové úrovni pro platformu Xilinx ZYNQ

Jiří Husák*



Abstrakt
Počítačové systémy se stávají stále složitější. V oblasti zpracování videa jsou potřeba stále vyšší nároky na výpočetní výkon. K těmto účelům lze použít technologii FPGA, která přináší možnost akcelerace v hardwaru. Vyšší nároky jsou však kladeny i na vývojáře. Proto vzniká snaha popisovat chování hardwaru pomocí vyšších programovacích jazyků. Cílem této práce je navrhnout a implementovat aplikaci pro zpracování videa akcelerovanou v FPGA pomocí jazyka C++ a zhodnotit přínos tohoto přístupu pro vývojáře. Aplikace je určena pro platformu Xilinx ZYNQ a pro implementaci je použito vývojové prostředí Xilinx Vivado. V architektuře jsou používány komponenty vytvořené pomocí syntézy na systémové úrovni (High-Level Synthesis). Výsledkem práce je demonstrační aplikace, která ukazuje, že může být výhodné používat syntézu na systémové úrovni. Hlavními důvody je úspora času při vývoji a možnost snadněji provádět změny chování komponenty. Tato práce je užitečná pro vývojáře FPGA a předkládá jim možnost zvýšení produktivity.

Klíčová slova: High-Level Synthesis — Xilinx ZYNQ — Xilinx Vivado — zpracování obrazu

Přiložené materiály: Demonstrační video — Stažení zdrojových kódů

*xhusak05@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

[Motivace] S rozvojem elektroniky na trhu rostou rovněž nároky na vývojáře. Velikost integrovaných obvodů u nových produktů se stále zvětšuje a také se zvyšuje celková složitost systému. Tato práce ukazuje možnost urychlení vývoje aplikací pro zpracování videa v FPGA za pomoci vyššího programovacího jazyka. Požadavkem na aplikaci je, aby bylo zpracováno video v reálném čase a aby vývoj nebyl časově náročný.

[Definice problému] Cílem práce je navrhnout a implementovat aplikaci pro zpracování videa. V průmyslu je často potřeba v reálném čase zpracovávat video s velkým rozlišením. Vhodnou platformou pro řešení těchto problémů je systém na čipu Xilinx ZYNQ. ZYNQ obsahuje dvoujádrový procesor ARM Cortex A9 a programovatelnou logiku FPGA. Výhodnou je možnost rozdělení aplikace na softwarovou a hardwarovou část. Aplikace bude zpracovávat video v HD rozlišení s frekvencí 60 snímků/s. Vstupní obraz bude filtrován (detekce hran, odstranění šumu, vyhlazení obrazu) nebo bude spuštěna detekce objektu v reálném čase. Výsledek bude zobrazován na HDMI monitoru.

[Existující řešení] V dnešní době se pro zpracování videa často používá softwarové řešení na běžném

CPU. Nevýhodou může být velký příkon a nedostatečná rychlost zpracování. Rychlost zpracování lze akcelerovat na GPU. Nicméně problém s příkonem roste. Vhodným řešením může být technologie FPGA. Tento přístup řeší jak výkonnost, tak příkon. Pro akceleraci zpracování videa v reálném čase na FPGA existují různá řešení [1, 2]. Rovněž platforma Xilinx ZYNQ se používá pro akceleraci zpracování videa v dopravních aplikacích [3, 4].

FPGA se v současnosti typicky programují v HDL jazycích, což ovšem přináší nevýhodu - časovou náročnost při vývoji. Proto se v posledních letech začíná uplatňovat přístup, kdy je chování obvodu popsáno pomocí vyšších programovacích jazyků [5].

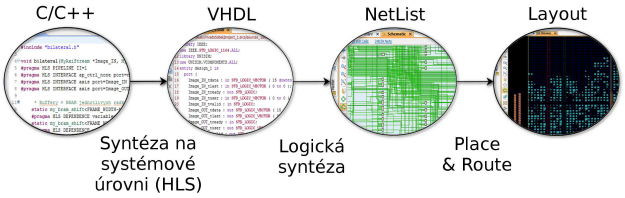
[Navržené řešení] Hlavní změnou oproti běžné implementaci pomocí HDL jazyků bude použití syntézy na systémové úrovni při implementaci komponent v FPGA. Komponenty jsou psány v jazyce C++ a syntetizovány pomocí vývojového prostředí Xilinx Vivado HLS. V FPGA budou použity i části psané ve VHDL a vestavěné IP-Cores od výrobce, se kterými budou komponenty propojeny.

[Přínos] Tato práce ukazuje praktické využití syntézy na systémové úrovni v reálné aplikaci. Práce má za cíl vyzkoušet a zhodnotit nový přístup k programování FPGA, který přispívá k zvýšení produktivity.

2. Teorie

[Syntéza na systémové úrovni] Cílem syntézy na systémové úrovni (High-Level Sysnthesis) je nalezнout RTL realizaci číslicového obvodu, který je popsán ve vysokoúrovňovém jazyce (C/C++, SystemC, ...). Obecně výsledných realizací může existovat více. Syntéza však hledá optimální realizaci vzhledem ke vstupním omezením (frekvence, latence, množství zdrojů). Syntéza pracuje v několika krocích. Nejdříve se vstupní algoritmus převede na Control / Data Flow Graph. Následně se alokují zdroje, které jsou potřeba. Poté probíhá plánování operací do jednotlivých taktů. Další fází je přiřazení funkčních jednotek k jednotlivým operacím. Poslední fází je vygenerování RTL a vytvoření konečného automatu obvodu. Výsledek může být například v jazyce VHDL nebo Verilog [6]. Na obrázku 1 můžeme vidět, že z RTL specifikace lze pomocí logické syntézy a Place & Route vygenerovat výslednou konfiguraci FPGA čipu (Layout).

Hlavní optimalizace, které se používají při návrhu obvodu pomocí syntézy na systémové úrovni se týkají proměnných, smyček a funkcí. U proměnných lze nastavovat bitovou délku, smyčky lze rozbalovat, slučovat a nebo zřetězit.



Obrázek 1. Vývojový diagram při navrhování FPGA za použití syntézy na systémové úrovni.

[Platforma Xilinx ZYNQ] Systém na čipu ZYNQ od firmy Xilinx obsahuje aplikační procesor ARM Cortex A9 a také programovatelné hradlové pole. K procesoru ARM jsou připojeny periferie Ethernet, USB, UART, SD karta a další. Můžeme zde také spustit operační systém Linux [7].

Programovatelné hradlové pole se skládá z matice CLB (Configurable Logic Blocks), z bloků DSP a BRAM a I/O rozhraní. Každá CLB se skládá z několika prvků LUT (Lookup Table) a prvků FF (Flip-flop). LUT lze použít jako logickou funkci, jako posuvný registr nebo jako paměť. FF se používá jako jednobitový registr. BRAM slouží jako paměť pro větší bloky, jedna BRAM má velikosti 36Kb a disponuje dvěma porty. BRAM se používá k implementaci pamětí FIFO, RAM a ROM. Komponenta DSP48 slouží pro rychlé aritmetické operace, kde operandy mají větší datovou šířku.

Po provedení syntézy nám nástroj sdělí, kolik prvků jednotlivých komponent algoritmus potřebuje k implementaci obvodu. Například varianta čipu ZYNQ Z-7020 obsahuje 106 400 FF, 53 200 LUT, 140 BRAM a 220 DSP48.

Komunikace uvnitř systému na čipu ZYNQ používá standard AXI. Pro vysokorychlostních přenos streamovaných dat se používá protokol AXI4-Stream. Pro mapování proměnných do paměti lze použít protokol AXI4-Lite. K proměnné má přístup jak aplikační procesor, tak FPGA.

[Vývojové prostředí Vivado] Prostředí Vivado Design Suite obsahuje několik aplikací. V této práci se budeme soustředit na aplikaci Vivado HLS, která z jazyka C/C++/SystemC syntetizuje algoritmus do HDL popisu obvodu v jazyce Verilog nebo VHDL. Slouží pro vytváření komponent (IP jader), které lze vkládat do celkového návrhu v aplikaci Vivado. Komponenty lze propojovat s vestavěnými IP jádry od výrobce nebo s vlastními jádry popsanými v HDL jazyce [8].

Při projení komponent je třeba správně definovat její rozhraní. Lze použít klasického drátového rozhraní nebo využít komunikačního protokolu. U drátového spojení lze použít valid signál nebo použít signály pro handshake protokol. Další možností je vytvořit

rozhraní typu FIFO, memory, bram, AXI4-Stream nebo AXI4-Lite.

Pro optimalizaci kódu v prostředí Vivado HLS se používají TCL skripty nebo se mohou psát direktivy pro syntézu do zdrojového kódu (#pragma HLS). Vivado HLS obsahuje debugger kódu. Po dokončení syntézy lze analyzovat, jaké operace jsou naplánovány do jednotlivých taktů. Zobrazí se informace o latenci, propustnosti a odhad použitých zdrojů.

Při implementaci komponent lze využít vestavěné knihovny (HLS C Libraries), která rozšiřuje jazyk C++ o datové typy a třídy usnadňující popis hardwaru. V knihovně jsou třídy pro bitové datové typy, matematické funkce s různou přesností upravené pro hardware, podpora práce s posuvnými registry nebo streamovanými daty.

3. Návrh aplikace

Aplikace bude přijímat obraz z kamery nebo ze statického obrázku a bude jej zpracovávat v FPGA. Výsledek bude zobrazen 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 bude aplikace obsahovat AdaBoost detektor SPZ.

[Hardware k aplikaci] Aplikace bude spuštěna na vývojové desce Xilinx ZC702, která obsahuje čip ZYNQ Z-7020. Deska obsahuje 1 GB paměti RAM. Ke kitu bude připojena přes rozhraní Ethernet kamera Unicam M621 z firmy CAMEA. Kamera pracuje v 8-bit režimu ve stupních šedi. Rozlišení kamery je 752 * 478 pixelů a frekvence 60 snímků/s a používá UDP protokol [9]. Aplikace bude pracovat s HD rozlišením (1280 * 720) a takový obraz se bude zobrazovat na připojeném HDMI monitoru.

[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. Rovněž již 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 detektoru AdaBoost jsem spolupracoval s Ing. Filipem Kadlčkem z firmy CAMEA. Získal jsem natrénovaný klasifikátor AdaBoost na SPZ [10]. Také jsem obdržel popis architektury, která bude implementována pomocí prostředí Vivado HLS.

[Návrh řešení aplikace] 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 umožňuje také načíst do paměti statický obrázek.

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.

[Komponenty řízení videa] 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] Sobelův filtr používá okno 3x3 pixelů a počítá s detekční maskou v ose x a y. Poté je hodnota zprůměrována. Filtr také ošetřuje okrajové hodnoty a používá nejbližší platnou hodnotu.

Mediánový filtr, který odstraňuje šum v obraze pracuje s oknem 5x5 pixelů. Porovnávají se sousední hodnoty a medián se spočítá po 24 krocích.

Bilaterální filtr vyhlazuje obraz a nechává ostré přechody. Pracuje s oknem 11x11 pixelů. Pokud je velký rozdíl mezi pixely, tak hranu zanechá. Postupně pro menší změny v obraze výsledný obraz vyhlazuje.

[Komponenty architektury klasifikátoru] Architektura klasifikátoru je určena pro nalezení jedné SPZ v obraze. Kvůli ušetření zdrojů je použit algoritmus AdaBoost s 50 slabými klasifikátory (LBP). Jelikož samotný klasifikátor používá rozhraní datových signálů, je nejprve video AXI-Stream převedeno na signály pomocí komponenty Axis2Wire. Poté následuje jednotka měnící rozlišení obrazu o pětinu. Komponenta se jmenuje ImageScale. Změna rozlišení se provádí metodou nejbližšího souseda. Z této komponenty vstupuje video do klasifikátoru.

Na výstupu klasifikátoru je komponenta AdaBoostOut, která počítá aktuální řádek a sloupec. Také provádí prahování výstupu z klasifikátoru. Pokud je hodnota

větší jak práh, posílá ji do komponenty MaxAdaBoost. 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.

4. Implementace

Pro implementaci komponent v prostředí Vivado HLS lze zvolit jazyk C++. Pro nastavování chování syntézy se do jazyka C++ vkládají direktivy (#pragma HLS). Oproti návrhu na běžném CPU je třeba patřičně upravit zdrojový kód, aby byly dosaženy požadované cíle na hardwaru:

- Upravit bitovou šířku u proměnných (možnost pevné řádové čárky).
- Definovat typ rozhraní u I/O (signály, protokol, AXI standard).
- Definovat místo a způsob uložení dat (BRAM, registry, LUT, rozdělit dlouhá pole, ...)
- Optimalizovat smyčky a funkce (zřetězení, rozbalení, ...).

Dále budou představeny jednotlivé optimalizace, které byly provedeny při implementaci aplikace.

[BRAM jako FIFO u filtrů obrazu] Obrazový filtr v FPGA pracuje s proudem pixelů. Typicky každý takt přichází do linky jeden pixel a zároveň každý takt jeden pixel vystupuje. Filtr obrazu při počítání pixelu potřebuje mít přístup k okolním bodům. Na běžném CPU není problém mít v paměti celý snímek a libovolně přistupovat ke všem bodům. Na FPGA je však limitovaná paměť. Proto se ukládá jen ta část, která je nezbytně nutná pro výpočet filtru.

Obrázek 2. 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.

Na obrázku 2 je potřeba modré řádky uložit do BRAM a v jednom taktu přečíst a také zapsat jeden pixel. BRAM má 2 porty, takže tuto operaci lze provést. Je nutné ručně vypnout datovou závislost, protože je zajištěna podmínka čtení a zápisu z rozdílných míst. Maska filtru je uložena v registrech, protože je potřeba číst všechny hodnoty v jednom taktu a provést nad nimi výpočet. Tento způsob implementace je použit v Sobelově, mediánovém i bilaterálním filtru.

[Rozhraní AXI] V prostředí Vivado je mnoho komponent od výrobce s rozhraním AXI, rovněž předávání dat mezi ARM procesorem a FPGA je pomocí AXI protokolu. Proto je výhodné jej použít při tvorbě komponent. V prostředí Vivado HLS stačí deklarovat strukturu a přidat ji do parametrů top funkce a poté direktivou pro rozhraní nastavit jako AXI-Stream. Vivado HLS automaticky přidá valid a ack signál a zajistí handshake protokol. V C++ kódu se programátor už nemusí starat o řízení sběrnice, stačí jen zapsat nebo číst z proměnné. Podobně také funguje protokol AXI-Lite, který přidělí proměnné adresu v paměti a je možné k této proměnné přistupovat z FPGA komponenty, i z ARMu. V aplikaci je AXI-Steam použit pro video signál, AXI-Lite je použit například při nastavování jasu ve videu, kdy jej uživatel může měnit za běhu.

[Převod AXI-Stream na signály] V některých případech můžeme potřebovat připojit do návrhu komponentu třetí strany, která nepoužívá standard AXI, ale běžné signály s různou bitovou šířkou. Toho lze dosáhnou vytvořením jednoduché komponenty. Na vstupu je AXI-Stream a na výstupu datové signály s valid signálem. Valid signál se nastavuje automaticky na jeden takt, když se do proměnné zapíše. Tato komponenta je použita při připojení AdaBoost klasifikátoru.

[Zřetězení funkce] Častou optimalizací komponent je zřetězení operací (pipelining). Funkce přijímá a produkuje každý takt výsledek. Latence ale může být několik taktů a funkce má 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.

Ve Vivadu HLS lze provést zřetězení pomocí jedné direktivy. Aplikace automaticky hledá plán operací, aby byl splněn požadovaný interval. V praxi je často nutné ještě upravovat kód, aby bylo provedeno požadovaného zřetězení. Nesmí se například přistupovat ke statické proměnné nebo k rozhraním několikrát během výpočtu. Pokud je při zřetězení problém, syntéza napíše informace a snaží se najít nejnižší možný interval zřetězení. Zřetězení je použito v aplikaci v každé komponentě, jelikož se zpracovává proudové video.

[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 de-

setinné číslo v pevné řádové čárce. Samotná operace trvala 31 taktů.

Možnost, jak urychlit latenci je předpočítat si tabulku pro dělení. Dělitel v tomto případě je 10b 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.

[Pevná řádová čárka] Vivado HLS podporuje i počítání v plovoucí řádové čárce (float, double). Operace s takovými čísly zabírá ale mnoho zdrojů. Proto se často volí způsob uložení čísel do pevné řádové čárky (fixed point). Pomocí bitových posunů lze provádět operace s desetinnými čísly velmi efektivním způsobem.

[Simulace kódu v C/C++] Vivado HLS přináší možnost simulovat kód na CPU pomocí test-bench aplikace. Tento přístup přináší rychlejší testování navrhované komponenty. V test-bench lze využít i knihovnu OpenCV. Například je možné si načíst obrázek a spustit referenční filtr obrazu a poté upravený algoritmus pro hardware. Vizuálně lze porovnat výsledky obou implementací. Této možnosti jsem využíval u testování komponent pracujících s obrazem.

5. Vyhodnocení výsledků

V této kapitole jsou diskutovány výsledky implementace. Jsou zde také popsány výhody a úskalí použití syntézy na systémové úrovni.

Jednou z hlavních výhod syntézy na systémové úrovni je úspora času při implementaci komponent. Například jednoduché komponenty, které jsou použity pro řízení video signálu, jsou popsány v jazyce C++ velmi efektivně. Firma Xilinx prezentuje na svých stránkách dosažení až čtyřnásobné úspory času při vytváření komponent [11].

Jelikož bilaterální filtr implementoval Ing. Martin Musil v jazyce VHDL, měl jsem možnost porovnat toto řešení s implementací ve Vivadu HLS.

Tabulka 1. Porovnání Bilaterálního filtru - VHDL implementace a HLS

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

Přesto, že se implementace mírně lišily (například v HLS bylo použito přesnějšího dělení s tabulkou), tak podle tabulky 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.

Úspora času při vývoji se zejména projevila u jednodušších komponent pro řízení videa. V prostředí Vivado HLS šlo velmi efektivně vytvořit rozhraní AXI-Stream a AXI-Lite. Poté stačilo zapsat chování komponenty a syntéza automaticky provedla obsluhu těchto protokolů.

Další výhodou je rychlý průzkum mikroarchitektur. Vývojář si může snadno analyzovat řešení a optimalizovat ho na potřebná omezení. Například lze snadno zjistit počet zdrojů v závislosti na frekvenci u zřetězeného kódu bez modifikace vstupního algoritmu. Tabulka 2 ukazuje, že s rostoucí frekvencí obvodu roste také latence a počet zdrojů.

Tabulka 2. Závislost množství zdrojů na frekvenci u Sobelova filtru

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

Nevýhodou použití syntézy na systémové úrovni je nemožnost mít návrh plně pod kontrolou. Nelze explicitně přistupovat k času při popisu v jazyce C. Nelze také určit, které operace se provedou v daném taktu. Rovněž může být v některých případech omezující, že je potřeba více zdrojů.

6. Závěr

[Shrnutí článku] Tato práce se věnuje využití syntézy na systémové úrovni v reálné aplikaci. Čtenář se v teoretickém úvodu dozví o problematice syntézy na systémové úrovni při návrhu a implementaci FPGA na platformě Xilinx ZYNQ. V další kapitole jsou popsány jednotlivé části aplikace. V kapitole implementace jsou popsány optimalizace, které byly použity. Při zhodnocení výsledků jsou popsány klady a nedostatky při použití syntézy na systémové úrovni.

[Výsledek práce] Cílem práce bylo vytvořit aplikaci v prostředí Xilinx Vivado. Tento úkol se mi podařilo splnit, chování aplikace lze vidět v demonstračním videu. Ukázal jsem tak na demonstrační aplikaci, že lze používat syntézu na systémové úrovni

v reálných aplikacích a že pomáhá k efektivnějšímu způsobu programování FPGA.

[Přínos článku] Hlavním přínosem článku je získání zkušeností s vývojovým prostředím Xilinx Vivado HLS. Článek tak ukazuje vývojářům FPGA možnost využití jednoduššího a rychlejšího vývoje.

[Možnost pokračování] V budoucnu lze navázat na tuto práci a použít zkušenosti s prostředím Xilinx Vivado HLS při implementaci dalších aplikací. Kromě oblasti zpracování obrazu lze pracovat i v oblasti síťového provozu nebo zpracování signálů. Zajímavou prací by mohlo být komplexnější porovnání využitých zdrojů a času vývoje oproti klasické implementaci pomocí HDL jazyků nebo porovnání s technologií GPU.

Literatura

[1] Zemcik Pavel. Hardware acceleration of graphics and imaging algorithms using fpgas. ACM, 2002.

[2] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich. A comparison of embedded reconfigurable video-processing architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 587–590, 2008.

[3] Yan Han and E. Oruklu. Real-time traffic sign recognition based on zynq fpga and arm socs. In *Electro/Information Technology (EIT), 2014 IEEE International Conference on*, pages 373–376, June 2014.

[4] J.M. Borrmann, F. Haxel, D. Nienhuser, A. Viehl, J.M. Zollner, O. Bringmann, and W. Rosenstiel. Stellar - a case-study on systematically embedding a traffic light recognition. In *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on*, pages 1258–1265, Oct 2014.

[5] Sol Pedre, Tomáš Krajník, Elías Todorovich, and Patricia Borensztejn. Accelerating embedded image processing for real time: a case study. *Journal of Real-Time Image Processing*, pages 1–26, 2013.

[6] Elliott John P. *Understanding Behavioral Synthesis*. Kluwer Academic Publishers, 1999. ISBN:0-7923-8542-X.

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

[8] Xilinxt. Vivado design suite user guide high-level synthesis, 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug902-vivado-high-level-synthesis.pdf.

[9] Musil Martin. Přenosy rastrových dat v fpga. Master's thesis, VUT Brno FIT, 2012.

[10] Kadlček Filip. Implementace obrazových klasifikátorů v fpga. Master's thesis, VUT Brno FIT, 2010.

[11] Xilinx. Accelerating integration. http://www.xilinx.com/products/design-tools/vivado/integration.html.