VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

DEMONSTRACE VYUŽITÍ PLATFORMY SYSTEM ON CHIP PYNQ Z2

DEMONSTRATION USAGE OF SYSTEM ON CHIP PLATFORM PYNQ Z2

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PATRIK POLÁŠEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. LUKÁŠ KEKELY, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Polášek Patrik, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Vestavěné systémy
Název: **Demonstrace využití platformy System on Chip Pynq Z2**
Demonstration Examples for Pynq Z2 System on Chip Platfrom
Kategorie: Návrh číslicových systémů
Zadání:

1. Seznamte se s platformou Pynq Z2 osazenou čipem Xilinx Zynq typu System on Chip a jeho hlavními parametry. Zaměřte se na analýzu dostupných periferií, vlastností procesního systému a také dostupnou programovatelnou logiku.
2. Zvolte několik (alespoň 3 až 5) ukázkových aplikací, kterými by bylo možné vhodně demonstrovat vlastnosti a použití uvedené platformy. Nabízí se akcelerace zpracování dat z dostupných periferií pro přenos zvuku, videa nebo síťových dat.
3. Navrhněte způsob implementace a prezentace zvolených aplikací s ohledem na jejich použití jako pomůcky k výuce. Cílem je ulehčit seznámení studentů s programováním na platformě typu SoC a poskytnout jim odrazový můstek pro implementaci jejich specifické aplikace.
4. Vytvořte a odlaďte implementace navržených aplikací na cílové platformě. Ke každé aplikaci připravte také podrobnou dokumentaci s popisem postupu implementace.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kekely Lukáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 29. října 2021

Abstrakt

Práce se zabývá vývojovou platformou Pynq Z2 s SoC obsahujicim programovatelnou logiku FPGA propojenou s procesorem ARM. Hlavním cílem je vytvoření skupiny vzorových aplikací, které využívají periferie dostupné na vývojové desce a realizují kritické výpočty na FPGA. Tyto aplikace mají podobu šablony dělící funkcionalitu na část komunikující s periferií a druhou část implementující samotný algoritmus výpočtu. Zvoleny byly konkretní algoritmy z oblasti vyhledávání v textu (Knuth-Morris-Pratt algoritmus), filtrace obrazu (změna barev obrazu a vyhlazovací konvoluční maska), filtrace zvukového signálu (dolní propust) a klasifikace internetových paketů (rozhodovací strom). Algoritmy je možné nahradit za vlastní, přičemž okolní rozhraní pro komunikaci s periferií zůstane zachováno. Kromě samotné implementace je ke každé aplikaci poskytnut interaktivní Jupyter Notebook dokument s doprovodným materiálem, který má za cíl usnadnit pochopení dané problematiky.

Abstract

The thesis deals with the Pynq Z2 with SoC containing FPGA programmable logic connected to ARM processor. The main goal is to create a set of sample applications that use the peripherals available on the development board and perform critical computations on the FPGA. These applications take the form of a template dividing the functionality into a part communicating with the peripherals and another part implementing the actual computation algorithm. Specific algorithms were chosen from the areas of text search (Knuth-Morris-Pratt algorithm), image filtering (image color change and smoothing convolution mask), audio signal filtering (low pass), and internet packet classification (decision tree). The algorithms can be replaced with custom ones, while the surrounding interface for communication with the periphery is preserved. In addition to the implementation itself, an interactive Jupyter Notebook document is provided for each application with accompanying material to facilitate understanding of the subject matter.

Klíčová slova

Pynq Z2, FPGA, vyhledávání v textu, obrazový filtr, zvukový filtr, klasifikace internetových paketů, SoC, Zynq

Keywords

Pynq Z2, FPGA, text search, image filter, sound filter, internet packet classification, SoC, Zynq

Citace

POLÁŠEK, Patrik. *Demonstrace využití platformy System on Chip Pynq Z2*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Lukáš Kekely, Ph.D.

Demonstrace využití platformy System on Chip Pynq Z2

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Lukáše Kekelyho, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Patrik Polášek
17. května 2022

Poděkování

Rád bych poděkoval vedoucímu práce, panu Ing. Lukáši Kekelymu, Ph.D., za odborné vedení, konzultace a věcné připomínky, které vedli ke zkvalitnění obsahu práce.

Obsah

Kapitola 1

Úvod

Vývoj jednodeskových počítačů umožnil jejich nasazení i mimo čistě průmyslovou oblast. Výrobci rozšiřují svá řešení, nabízí lepší parametry a konkurují cenou. Relativně nízká cena a dostatečné schopnosti jednodeskových počítačů zvedly zájem kutilů. Zájemci o jednodeskové počítače přispívají novými nápady využití a dalšími modifikacemi jak po programové, tak po hardware stránce. Pozadu nezůstaly ani externí periferie, neustále se zvyšuje rozlišení obrazovek, roste objem přenášených dat na internetu a obecně jsou kladeny přísnější nároky na přenos a zpracování většího množství informací. Programové obsloužení časově kritických operací, mezi které se řadí zpracování videa, zvukové filtry nebo klasifikace internetových paketů, nemusí být dostačující. Jako řešení se nabízí přesun časově náročných operací do hardwarově akcelerovaných výpočtů.

Práce je zaměřena na platformu Pynq Z2. Základem této platformy je systém na čipu (SoC). Jde o čip z jednoho kusu křemíku, který má v sobě zabudovaný ARM procesor, řadič paměti a další moduly. Významným modulem je obvod programovatelné hradlové pole (FPGA). Výskyt FPGA v SoC není běžnou záležitostí a jde spíše o výjimku. FPGA obsahuje prvky k vytváření hardwarově akcelerovaných obvodů a lze jej opakovaně přeprogramovávat podle potřebné funkcionality. Tento čip umožňuje delegovat kritické úkoly z procesoru na specifický hardware akcelerátor a zlepšit efektivitu využití celé platformy.

Cílem této práce je seznámení s platformou Pynq Z2 a způsobem jejího programování. Představit možnosti a potenciál FPGA pro akceleraci zpracování dat z oblastí vyhledávání v textu, zpracování obrazu, filtrace zvukového signálu a klasifikace síťových paketů. Kromě teoretického popisu také prakticky a vhodným způsobem tyto schopnosti prezentovat na demonstračních aplikacích. Ke každé aplikaci je proto podrobně popsán postup jejího vytvoření, použití a rozšíření formou doprovodného manuálu v podobě interaktivního Jupyter Notebooks dokumentu. Tento dokument a ukázkové aplikace mohou posloužit všem zájemcům o danou problematiku i studentům, kteří se v budoucnu setkají s platformou Pynq Z2 na fakultě.

Následující text práce je členěn do 5 kapitol. Kapitola 2 je seznámením s platformou Pynq Z2. Popsán je zde přístup k SoC platformě, způsob programování i hierarchický náhled na komponenty důležité pro tuto práci. V kapitole 3 jsou představeny demonstrační aplikace, seznámení s algoritmy pro řešení daných problémů a naznačení akcelerace kritické části algoritmu v paralelním řešení na FPGA jednotce. Kapitola 4 obsahuje návrhy jednotlivých aplikací a detailnější pohled na rozbor problémů, které je třeba řešit. Realizace aplikací s implementačními detaily, které se vážou na platformu Pynq Z2, se nachází v kapitole 5. Závěrečná kapitola 6 shrnuje výsledky práce a nabízí směr, kterým ji lze dále rozvíjet.

Kapitola 2

Platforma systém na čipu Pynq Z2

Obsahem této kapitoly je úvod do problematiky systému na čipu (SoC) a souvislost s platformou Pynq Z2. Část kapitoly je věnována pohledu na platformu Pynq Z2 jako na ucelený systém. Dále jsou v jednotlivých sekcích rozebrány dílčí prvky SoC jako samostatné jednotky a jejich návaznost na systém jako celek.

Následující 3 odstavce o SoC vychází z literatury [5]. Pokročilé výrobní technologie umožnily zmenšit a vrstvit na sebe tranzisotry a rezistory. Dostatečné množství těchto hardwarových prvků na jednom substrátu křemíku slouží k vytvoření SoC pro aplikačně specifické nasazení. Hlavním úkolem při vytváření SoC je nalezení optimálního vyvážení ceny, výkonu a požadavků na specifickou aplikaci.

Úkolem návrhu SoC je vhodně rozložit implementaci funkcí mezi programové řešení na obecném procesoru a hardware řešení v podobě aplikačně specifického obvodu. Kompromis závisí na konkrétní aplikaci a využití SoC. Programová implementace na obecném procesoru je pomalejší a energeticky náročnější, ale zato přináší výhodu v podobě snadného ladění aplikace a flexibilní možnost úpravy aplikace přeprogramováním funkce. Hardware implementace funkce je rychlá a energeticky efektivnější, ale je implementována napevno. Změna funkce vyžaduje provést nový návrh hardware a ladění aplikace je rovněž komplikovanější než u programové varianty.

Na jedné straně se nachází čistě programové řešení na obecném procesoru. Na straně druhé je čistě hardware řešení v navrhnutém aplikačně specifickém integrovaném obvodu (ASIC). Odtud vychází závislost na cílové aplikaci a kritické funkce se převádí do hardware, zatímco je snahou ponechat programátorovi dostatečnou svobodu v modifikování aplikace a udržet cenu na rozumné úrovni. Systém na čipu se snaží vzít to lepší z obou krajních přístupů.

Možnost zahrnout velké množství komponent na malou plochu jednoho kusu křemíku se osvědčila. Většina vývojových plaforem používá jako základ SoC, který je zakomponovaný do desky plošných spojů (DPS). Na desce jsou další komponenty a periferie utvářející složitý systém – platformu připravenou k programování a zasazení do aplikace. Současně dostupné vývojové platformy jsou:

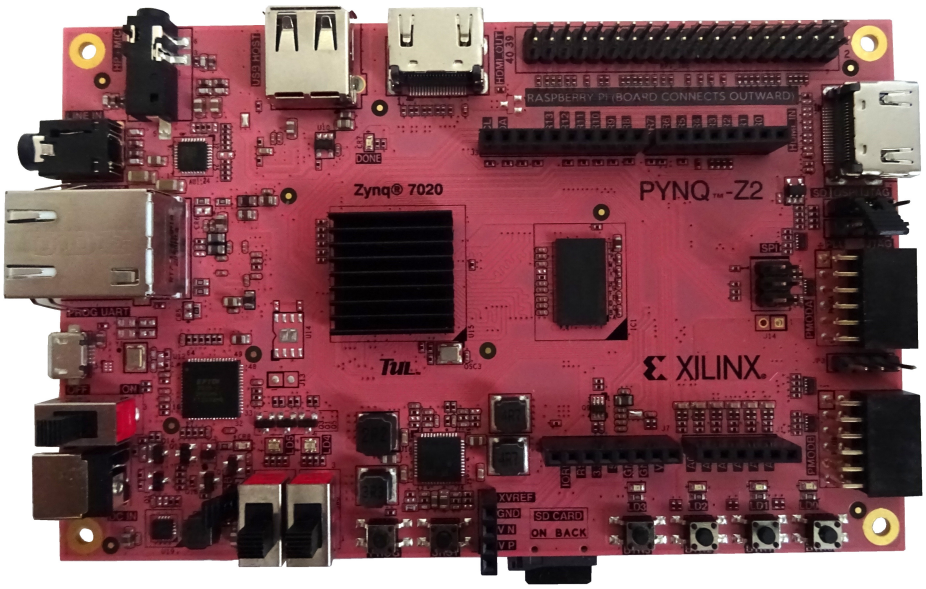
Espressif [4] – Společnost Espressif vyrábí vývojové desky se svým SoC řešením. Systémy na čipu jsou zaměřeny na bezdrátové komunikace Bluetooth a Wi-Fi. Uplatnění nacházejí zejména v Internetu Věcí (IoT) a chytrých domácnostech. Některé vývojové desky jsou opatřeny integrovaným akcelerátorem pro rozpoznávání hlasu. Použitý hardware nemá dostatečné zdroje pro běh běžného operačního systému a používá se FreeRTOS s rozšiřujícími knihovnami.

Intel NUC [7] – Intel NUC jsou mini počítače s operačním systémem Windows. Jejich SoC vychází ze standardních procesorů. Rozdílem je integrace grafického čipu, paměťového řadiče, USB řadiče, obvodu pro napájení a bezdrátových technologií na jeden čip, zatímco u běžných procesorů tyto záležitosti řešila samostatná čipová sada.

Raspberry Pi 4 [14] – Multifunkční platforma, která má dostatek zdrojů pro běh operačního systému Linux. K vývojové desce se dají připojit běžné periferie jako k PC. Na SoC jsou integrovány řadiče k rozhraním pro připojení periferií, navíc je dostupná sada 40 vstupně-výstupních pinů pro obecné využití. Nechybí integrovaný grafický akcelerátor nebo bezdrátové moduly pro Bluetooth a Wi-Fi komunikaci. K programování na vývojové desce se používá C++ nebo Python s rozšiřujícími knihovnami pro přístup k hardware pinům. Na Linux se dají doinstalovat i další překladače a využít ostatní programovací jazyky.

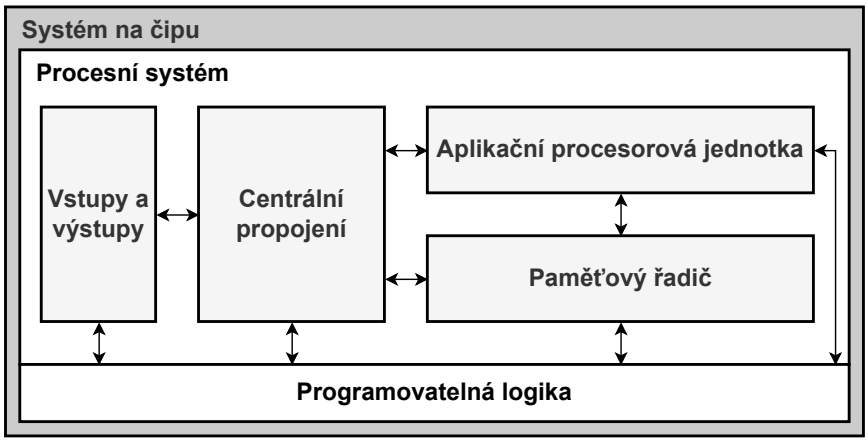
Arduino [1] – ekosystém Arduino zahrnuje vývojovou desku, překladač, tutoriály, technickou dokumentaci a různé projekty s příklady využití. Dostupných je několik řad vývojových desek, které se liší výkonem a počtem dostupných pinů k připojení externích periferií. Výkon není dostatečný pro běh operačního systému, nicméně postačuje pro velkou část projektů z oblasti vestavných systémů. Existuje celá řada kompatibilních periferí a modulů v podobě samostatných desek plošných spojů, které respektují rozložení pinů právě na Arduinu.

Pynq Z2 [12] – Všestranná vývojová deska s rozhraními běžnými pro počítače. Navíc jsou přítomné piny pro oblast vestavných systémů. Piny mají rozložení známá z vývojových desek Raspberry Pi a Arduino, což umožňuje použít a rozšířit již existující řešení. Pokročilý SoC s FPGA jednotkou pro vytváření hardware obvodů má dostatek výkonu pro běh operačního systému Linux. K programování je určen webový nástroj Jupyter Notebooks a programovací jazyk Python. Dostupné jsou všechny známé knihovny z Pythonu a pro vývojáře jsou přidány hardware knihovny použitelné na programování FPGA. Vývojovou desku Pynq Z2 lze vidět na obrázku 2.1.



Obrázek 2.1: Vývojová deska Pynq Z2.

Tato práce je zaměřena na platformu Pynq Z2, protože je bohatá na rozhraní pro externí periferie, kombinuje výhodu z Arduina a Raspbery Pi v podobě známého rozložení pinů a navíc obsahuje FPGA modul. Centrálním prvkem je čip Xilinx Zynq Z-7020 architektury SoC [23]. Struktura čipu je rozdělena na dvě hlavní části, procesní systém (PS) a programovatelnou logiku (PL). Rozdělení lze vidět na obrázku 2.2. Dále lze na obrázku vidět blok procesoru, řadiče paměti, vstupů a výstupů, což jsou typické části běžných výpočetních systémů. Hlavní odlišnost SoC od čipové sady je dána blokem interního propojení, který řídí komunikaci mezi všemi prvky a rovněž je umístěn na stejném čipu. Šipky znázorňují propojení bloků a směry, kterým mohou zahájit komunikaci.



Obrázek 2.2: Struktura čipu Xilinx Zynq Z-7020. Obrázek je zjednodušením blokového schématu z technické dokumentace [23].

Čip Zynq je příkladem SoC se zaměřením na aplikační využití. Časově nenáročné funkce lze implementovat na obecném procesoru. Flexibilita v podobě úpravy programu dovoluje funkci rychle upravovat a jednoduše ladit. Pro kritickou část je vyhrazena programovatelná logika FPGA. Funkce se navrhne jako implementace na úrovni hardwarové architektury obvodu. Vytvoří se konfigurační soubor obvodu, kterým se FPGA přeprogramuje. Návrh hardwarového obvodu je komplikovanější a trvá déle než programová realizace, avšak přináší výhody v nižší spotřebě, kratší latenci nebo vyšším výkonu dané aplikace.

Při porovnání FPGA s ASIC obvody je jednou z výhod možnost ladění a opravy chyb v návrhu. Vytvořený obvod se nemusí vyhazovat, pouze se opraví chyba a nahraje se nový konfigurační soubor. Pokud to aplikace dovoluje, dá se realizace v FPGA testovat přímo za běhu a použít rychlejší přístup k vývoji prototypováním. Další výhodou FPGA je znovuvyužití hardware prvků. Technologie i požadavky se mohou s časem vyvíjet (například přechod z IPv4 protokolu na IPv6) a starý návrh funkce se stává nepotřebným. ASIC obvod by se musel nahradit novým, FPGA obvod lze změnit a uspokojit nové požadavky.

Možnosti nasazení čipu Xilinx Zynq Z-7020 jsou široké a sahají do mnoha odvětví [23]:

- automobilový průmysl, podpůrný systém pro řidiče
- průmyslové řízení motorů, průmyslová komunikační síť a strojové vidění
- telekomunikace a rádiový signál
- chytré kamery komunikující po síti

- lékařská diagnostika a snímkování
- multifunkční tiskárny
- videozáznam a noční vidění

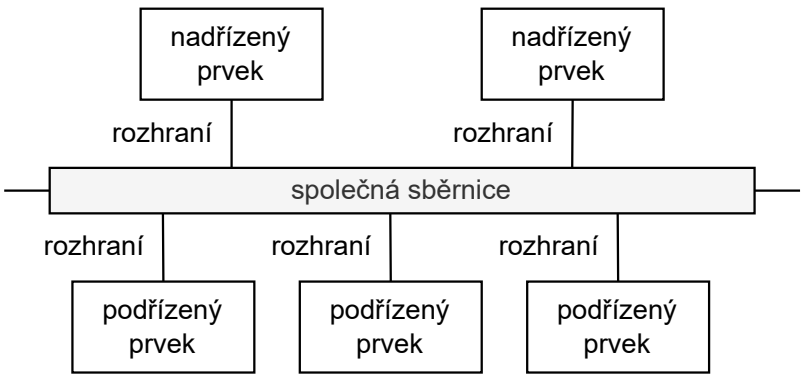
Propojení ve schématu 2.2 je založeno na AXI sběrnici, které se věnuje následující sekce. Další sekce se zabývají důležitými bloky a jejich úlohou v SoC.

2.1 AXI sběrnice

Architektura sběrnice sestává z pěti kanálů, tři pro zápis a dva pro čtení [8]. Zápisové kanály jsou adresa zápisu, zapisovaná data, potvrzení zápisu, čtecí kanály jsou adresa čtení a čtená data. Každý z těchto kanálů obsahuje pomocné signály *valid* a *ready* pro obousměrné potvrzování komunikace. Zdroj vysílající data vystaví signál valid v době, kdy jsou na sběrnici již platná data. Přijímající strana vystavuje signál ready v momentě, kdy je schopna přijímat data. Přenos dat probíhá až když jsou oba signály (read i valid) aktivní. Tímto mechanismem pomalejší prvek na sběrnici řídí rychlost komunikace. Kanály přenášející čtená nebo zapisovaná data mají navíc signál *last* indikující poslední položku přenosu.

Připojená zařízení na sběrnici jsou dělena na vedoucí a podřízená. Způsob zapojení prvků lze vidět na obrázku 2.3. AXI standard poskytuje jednotnou definici komunikačního rozhraní mezi vedoucím a podřízeným prvkem, mezi vedoucím prvkem a vnitřním propojením sběrnice a mezi podřízeným prvkem a vnitřním propojením sběrnice. Společné propojení lze realizovat třemi způsoby:

- sdílená adresová i datová sběrnice
- sdílená adresová a více datových sbernic
- více adresových i datových sběrnic



Obrázek 2.3: AXI sběrnice s připojenými prkvy.

Jednotlivé části systému jsou připojovány na sběrnici přes rozhraní. Zavedena byla dvě rozhraní. AXI-Lite je pro jednodušší řízení podobné registrovému přístupu, které neposkytuje plnou funkcionalitu AXI. Omezení spojená s AXI-Lite jsou:

- Přenosy nepodporují zřetězené přenosy, každá transakce může mít pouze jednu datovou položku.

- Všechny datové přístupy jsou zarovnány na šířku sběrnice, která je 32 nebo 64 bitů.
- Přístupy na sběrnici se nedají upravovat a ani ukládat do nějaké vyrovnávací paměti.
- Podporován není ani výlučný přístup.
- Transakce musejí probíhat v takovém pořadí, v jakém se budou vykonávat.

Druhým rozhraním je AXI-stream, které po ustálení komunikace (valid a ready signál) umožňuje přenášet libovolný počet bajtů. Přenosy jsou typicky organizovány do paketů, přičemž jeden paket je sestaven z více bajtů. Bajty mohou mít jeden ze tří významů. Datové bajty nesou přenášená data. Poziční bajty nesou informaci o relativní pozici bajtů v přenosu. Posledním typem jsou null bajty, které nenesou žádnou důležitou informaci, z přenosu mohou být klidně vynechány.

Mezi významné funkce AXI-stream rozhraní se řadí zmenšování komunikace vypouštěním null bajtů (angl. packing), převod komunikace z širší sběrnice na užší (například z 32 bitů na 16 bitů) a rozšiřování komunikace z menší sběrnice na širší (například z 16 bitů na 32 bitů). Proudové (angl. stream) přenosy mohou být i prokládány a rozhraní se postará o jejich seřazení.

2.2 Aplikační procesorová jednotka

Aplikační procesorová jednotka (APU) [23] v sobě zapouzdřuje ARM procesor rodiny Cortex-A9, která je nasazována v aplikačně zaměřených systémech. Procesor je postaven na architektuře RISC (z angl. Reduced Instruction Set Computer) a má dvě fyzická jádra. Základní vlastnosti procesoru jsou:

- Instrukční sada ARM s podporou Thumb-2.
- SIMD koprocesor se 128 bity pro vektorové zpracování dat.
- Oddělená paměť cache L1 se 32 KB pro instrukce a 32 KB pro data.
- Společná paměť cache L2 s 512 KB.

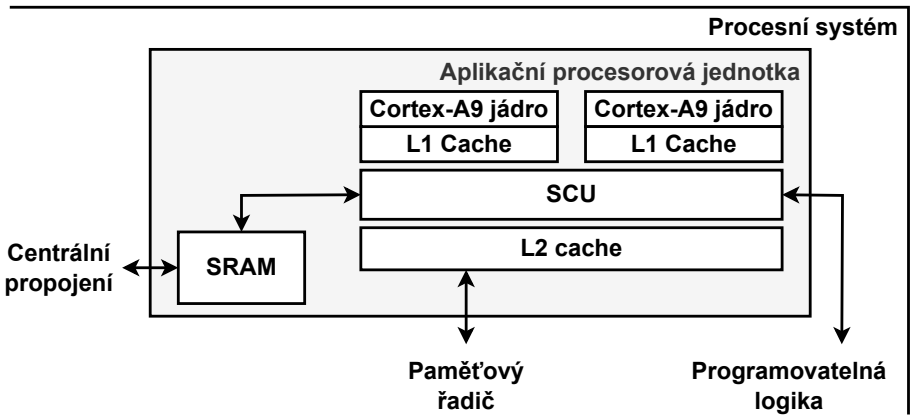
Instrukční sada ARM dodržuje jednotnou délku instrukcí, a to 32 bitů. Rozšíření Thumb-2 povoluje i 16 bitovou délku instrukcí, což vede na šetření místa v paměti instrukcí. Případně je omezeno čtení z paměti instrukcí, když se jedním přečtením načtou dvě 16 bitové instrukce. Jednotná délka instrukcí vede na jednodušší dekodér instrukcí, který může pracovat rychleji a efektivněji.

Procesor může vykonávat až 2 instrukce v jednom hodinovém cyklu, umožňuje zpracování instrukcí mimo pořadí, predikci skoků a rozbalování cyklů na HW úrovni. Predikce skoků je vyhodnocována jak staticky, při překladu programu, tak dynamicky za běhu vykonávání instrukcí. Prediktor si v BTAC (z angl. Branch Target Address Cache) ukládá dekódované adresy nedávných skoků i návratové adresy z podprogramů.

Funkcionalitu samotného procesoru rozšiřuje koprocesor NEON, který provádí vektorové zpracování dat. Podporované datové typy vektorového zpracování jsou znaménková i bezznaménková celá čísla (integer) v rozsahu 8, 16, 32 nebo 64 bitů na jedno číslo. Druhým podporovaným typem jsou desetinná čísla (float) se 32 bity na číslo. Poskytované matematické operace vektorové SIMD jednotky jsou sčítání a odčítání, násobení s volitelnou

akumulací součinů do jednoho výsledku, výpočet minimální nebo maximální hodnoty a výpočet převrácené hodnoty odmocniny z čísla ($\frac{1}{\sqrt{x}}$).

Samotné provádění výpočtů je nedostačující, procesor musí operovat nad daty. V komplexu aplikační procesorové jednotky je statická paměť RAM (SRAM) o velikosti 256 KB a paralelně vedle ní se nachází 512 KB L2 cache paměť. Umístění obou pamětí lze vidět na blokovém schématu 2.4. Komunikace z procesoru do pamětí je vedena přes speciální blok SCU (angl. Snoop Control Unit). Tento blok řídí prioritu komunikace a paměťových přenosů. Stará se o koherenci dat pamětí cache L1 a cache L2. Zajímavou vlastností SCU bloku je schopnost kopírovat data z paměti cache jednoho jádra do paměti cache druhého jádra bez nutnosti přístupu do hlavní paměti RAM a vyhnout se tak zdlouhavému přístupu.



Obrázek 2.4: Blokové schéma aplikační procesorové jednotky.

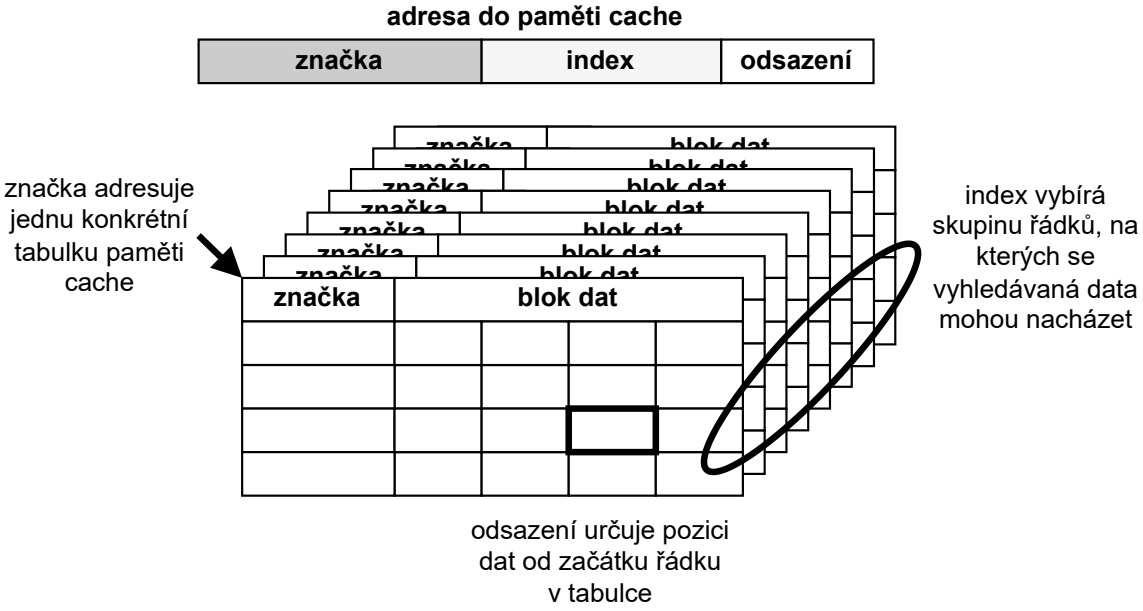
Dalším důležitým bodem je komunikace s externí pamětí RAM. Z obrázku 2.4 lze vypozorovat propojení komunikace mezi paměťovým řadičem a pamětí cache L2. Tento dedikovaný propoj jde mimo centrální propojení na SoC a činí paměť cache L2 významným prvkem přísunu dat ke zpracování do procesoru. Následující sekce je věnována principu fungování paměti cache a řadiči paměti.

2.3 Princip paměti cache a řadič externí RAM paměti

Paměť cache zastupuje důležitou úlohu jako vyrovnávací paměť. Procesor je schopen vykonávat instrukce řádově rychleji, než jak dokáže hlavní paměť RAM poskytovat data. Pokud procesor nemá včas data, nemá nad čím operovat a vznikají negativní prodlevy bez produkování užitečných výpočtů. Data se v cache uchovávají s ohledem na časovou a prostorovou lokalitu. Časová lokalita předpokládá, že pokud byla data použita nedávno, tak nejspíš budou vyžadována i v blízké budoucnosti. Prostorová lokalita předpokládá, že pokud byla použita data z nějakého paměťového místa, tak v budoucnu budou vyžadována sousední data z vedlejší adresy.

V katalogovém listu čipu Zynq [21] je uvedena 4 cestná skupinově-asociativní paměť cache L1 a 8 cestná skupinově-asociativní paměť cache L2. Počet cest znamená, na kolik tabulek je paměť rozdělena. Při zápisu nebo čtení se používá mapovací adresa se třemi složkami – značka, index a odsazení. Formát adresy s ukázkou adresace 8 cestné paměti je na obrázku 2.5. Index adresuje řádek v tabulkách, na kterém se mohou data nacházet. Tabulek je více podle počtu cest a značkou v adrese je nutné vybrat jednu odpovídající

tabulku. Poslední část z adresy je odsazení určující pozici vyhledávaných dat od začátku řádku v cache.



Obrázek 2.5: Formát adresy a příklad adresace v 8 cestné skupinově-asociativní cache.

Skupinově asociativní cache poskytuje kompromis mezi rychlostí přístupu k datům a obsazeností paměti. Data z hlavní paměti RAM se mohou namapovat na stejný řádek v paměti cache. U přímo mapované cache (pouze 1 cesta) by data mapována do jednoho řádku byla neustále nahrazována a zbytek řádků paměti by bylo nevyužito. Vyhledání dat je okamžité, protože mohou být pouze na jednom konkrétním řádku. Plně asociativní paměť umožňuje uložit data kamkoliv do paměti cache, při vyhledávání se musí projít všechny řádky v cache. U skupinově asociativní paměti (s 8 cestami) se data namapují na jeden řádek, a pokud je obsazený na jedné cestě, využije se jiná cesta. Při vyhledávání dat stačí najít konkrétní skupinu řádků a prohledat ji na 8 cestách.

Z paměti cache vede dedikovaný spoj, který opouští aplikační procesorovou jednotku, a míří do řadiče externí paměti RAM [23]. Řadič podporuje paměť typu DDR (Double Data Rate) verze 3 s kapacitou 512 MB, která přenáší data na vzestupné i sestupné hraně hodinového signálu.

AXI rozhraní z paměti cache L2 není jediné, které vede do paměťového řadiče. Další dvě 64 bitové AXI rozhraní vedou na HP (z angl. High Performance) porty programovatelné logiky. Poslední AXI rozhraní vede do centrálního propojení SoC a umožňuje klást požadavky na přístup do paměti všem ostatním zařízením, která potřebují komunikovat s hlavní operační pamětí.

Každé ze 4 rozhraní má dvě vyrovnávací paměti (FIFO fronty) pro požadavky na zápis a čtení. Řadič paměti se stará o prioritní výběr zpracování požadavků, který je založen na algoritmu round robin s přihlédnutím ke stárnutí požadavků. Požadavky jsou cyklicky vybírány od všech zařízení. Pořadí v cyklu se odvíjí od doby strávené čekáním na požadavek, vážností požadavku a informace, zda se požadavek nachází ve stejné paměťové stránce jako předchozí požadavek.

2.4 Programovatelná logika

Prvotní spuštění programovatelné logiky (FPGA) provází 4 fáze [23]:

1. Start-up fáze zajišťuje stabilizaci napájení.
2. Následně se nuluje statická RAM v programovatlené logice, připravuje se tak na konfiguraci.
3. Provede se konfigurace programovatelné logiky nahráním bitstream konfigurace do statické RAM.
4. Aktivuje se komunikační rozhraní mezi procesním systémem a programovatelnou logikou.

Konfigurace programovatelné logiky se provádí daty označovanými jako bitstream. Bitstream je uložen ve volatilní statické RAM paměti, z čehož vyplývá, že po vypnutí nebo restartu programovatelné logiky se data z paměti ztrácí a je potřeba nahrát konfiguraci znovu. Změnu funkce je možné provést i později částečným přeprogramováním, když je potřeba upravit nějaké koeficienty nebo změnit aktuální algoritmus v FPGA. [23]

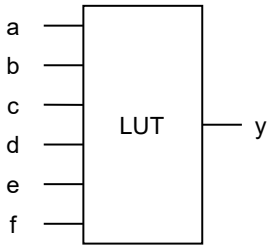
Přínosem programovatelné logiky je převedení softwarových algoritmů do paralelní podoby prováděné v hardware. Některé problémy tak lze řešit výrazně rychleji a s energetickou úsporou oproti sekvenčnímu provádění na procesoru s paralelizací dostupnou maximálně na úrovni počtu jader procesoru. Také se zvyšuje využitelnost architektury. Pokud algoritmus používá pouze některé operace, tak lze v hardware implementovat dostatečný počet bloků realizujících potřebnou funkci a využít tak většinu dostupných zdrojů. Zatímco obecný procesor má k dispozici velmi omezený počet hardware zdrojů a málokterý algoritmus využívá všechny možné funkce obecného procesoru současně. Naopak často zůstává velká část funkcí a zdrojů nevyužitých.

Podobný přínos mají i aplikačně specifické integrované obvody (ASIC), které jsou vyráběny přímo na křemíkovém čipu a z pohledu spotřeby a výkonu jsou ještě výhodnější než FPGA. Značnou nevýhodou je nemožnost upravit funkci obvodu a nutnost vytvořit nový návrh, který se musí verifikovat a otestovat, a až poté vyrobit nový obvod. FPGA lze snadno přeprogramovat a doplnit nové funkce (například k IPv4 protokolu přidat podporu pro IPv6, dolno propustní filtr upravit na horní propust atd.).

Hlavní součásti programovatelné logiky jsou [23]:

- Konfigurovatelné logické bloky (CLB) – obsahuje 6 vstupé look-up tabulky (LUT, celkem dostupných 53 200), registr a kaskádovou sčítačku (s přenosem přetečení vyššího řádu). Princip LUT tabulky ukazuje obrázek 2.6. Vstupy proměnných fungují jako adresa do vyhledávací tabulky. Na dané adrese je uložen výsledek funkce. Výsledky v tabulce lze programově upravit a vytvářet tak libovolné logické funkce pro až 6 vstupních proměnných.
- Blokové RAM s 36 kB paměti (celkem dostupných 140) – pro přístup do blokové RAM je použit duální port, volitelně lze paměť rozdělit a využít ji jako dvě 18 kB paměti.
- Digitální signálové procesor (celkem dostupných 220) – jde o procesor optimalizovaný na zpracování signálů a použití ve filtrech. Obsahuje 25×18 bitovou násobičku a následně 48 bitovou sčítačku, ve které je schopen akumulovat výsledky.

- Rozvod hodinového signálu s vysokorychlostními vyrovnávacími pamětmi (buffer), se schopností posunu fáze hodinového signálu i s filtrací zákmitů.
- Nastavitelné vstupně výstupní porty.



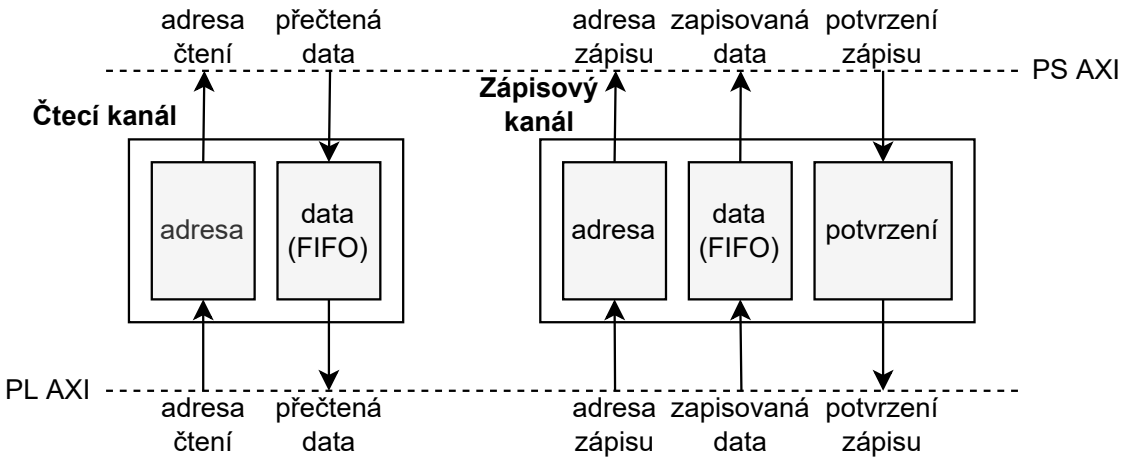
a	b	c	d	e	f	y
0	0	0	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	1
...	...	...	...	...	...	...
1	1	1	1	1	1	0

Obrázek 2.6: Vyhledávací LUT blok s ukázkou části tabulky pro kombinace vstupních proměnných a naprogramovanými výstupy.

2.5 Centrální propojení a periferie

Centrální propojení má podobu přepínačů point-to-point spojení. Řídí komunikaci mezi komponentami na SoC (pokud probíhá přes centrální propojení) a tvoří komunikační bránu mezi SoC a vstupními a výstupní porty, na které jsou připojeny ostatní periferie vývojové desky. Z programovatelné logiky na čipu Zynq vedou tři typy rozhraní, které jsou určeny ke komunikaci mezi periferiemi v procesním systému a programovatelnou logikou FPGA:

- AXI ACP (Accelerator Coherency Port) – jde o jedno AXI rozhranní, které spojuje programovatelnou logiku a jednotku SCU (Snoop Control Unit) nacházející se v aplikační procesorové jednotce (APU). SCU jednotka zastřešuje komunikaci paměti cache L2 a udržuje ji koherentní s pamětí cache L1 v procesorech. Přední vlastností AXI ACP portu je nízké zpoždění komunikace, která umožňuje pracovat s pamětí cache L2 stejným způsobem jako procesor. Postup akcelerace je takový, že procesor připraví data do paměti cache L2, programová logika si je převezme skrze AXI ACP port, data zpracuje a výsledek znovu skrze AXI ACP port nahraje do paměti cache, kde si výsledek převezme procesor.
- AXI HP (High Performance) – dostupná jsou čtyři AXI rozhraní, přičemž každé z nich má dvě vyrovnávací paměti v podobě FIFO fronty, jednu pro čtecí přenosy a druhou pro zápisové přenosy. Rozhraní vedou do řadiče paměťového propojení, které propojuje komunikaci mezi čtyřmi AXI rozhraními na straně programovatelné logiky a dvěmi AXI rozhraními vedoucími do řadiče paměti RAM nebo jedním AXI rozhraním vedoucím do SRAM v aplikační procesorové jednotce. Blokové schéma s čtecím a zápisovým kanálem jednoho AXI rozhraní HP portu lze vidět na obrázku 2.7.
- AXI GP (General Purpose) – tyto porty jsou napojeny přímo do bloku centrálního propojení. Tyto porty neobsahují žádné vyrovnávací paměti a jsou určeny pouze k obecným účelům a ne pro výkonově náročné aplikace.



Obrázek 2.7: Blokové schéma rozhraní AXI HP portu.[23].

Vzhledem k náplni této práce jsou nejdůležitější periferie a konektory pro přenos videa, zvuku a síťových dat. Pro přenos videa jsou na desce dva HDMI porty, jeden jako vstupní, druhý jako výstupní. Konektory na desce žádný svůj řadič nemají. Přenosy dat musejí být řízeny z programovatelné logiky.

Pro přenos zvuku jsou poskytnuty dva 3,5 mm Jack konektory, jeden pro vstupní audio signál, druhý pro výstup audio signálu. U výstupního konektoru se nachází 24 bitový digitálně-analogový převodník. Přenos dat je zarovnáván po 32 bitech, horních 8 bitů je nevyužito při dalším procesu zpracování je nutné horních 8 bitů odseknout.

Internetová data jsou přenášena prostřednictvím RJ-45 konektoru. O funkčnost se stará ethernetový řadič se standardem IEEE 802.3 podporující přenosy v rychlostech 10, 100 nebo 1000 Mb/s. Řadič v sobě implementuje řadu funkcí, například automatické zahazování chybných paketů, vypočítávání kontrolních součtů IPv4 i IPv6 paketů, generování přerušení při dokončených operacích odesílání a přijímání paketů.Seznam hlavních rozhraní a periferií [19] platformy Pynq Z2 je shrnut v tabulce 2.1.

USB a Ethernet
Gigabit Ethernet konektor
Micro USB-JTAG obvody
Micro USB-UART bridge
USB 2.0 (poze v roli host)

Přepínače, tlačítka a LED
4 tlačítka
2 přepínače
4 LED
2 RGB LED

Audio a video
Vstupní HDMI konektor
Výstupní HDMI konektor
Vstupní 3.5mm Jack konektor
Výstupní 3.5mm Jack konektor

Rozšiřující konektory
2 Pmod porty
Arduino Shield konektor
Raspberry Pi konektor

Tabulka 2.1: Přehledový seznam konektorů dostupných na vývojové desce Pynq Z2.

Kapitola 3

Využití hardware akcelerace

Tato kapitola představuje některé okruhy aplikací, na které se dá uplatnit hardware akcelerace. Akcelerace spočívá v paralelizaci operací, které jsou na běžném procesoru vykonávány sekvenčně. Každý z představených okruhů aplikací je zaměřen na trochu jiný typ dat. Za obecným popisem aplikačních oblastí následují sekce s jednotlivými problémy a algoritmy na jejich řešení.

První oblastí využití hardware akcelerace je vyhledávání řetězce v textu. Může se jednat o vyhledávání specifické fráze v úředních dokumentech nebo lékařských zprávách za účelem analýzy. Jako konkrétní příklad může být vyhledání počtu osob, které spáchali konkrétní trestný čin nebo vyhledání počtu osob, které se léčili s nějakou nemocí, případně kterým byl podán určitý lék. Data mining aplikace mohou získávat data z relevantních webových stránek, na kterých byla nalezena shoda s klíčovým řetězcem. Internetové vyhledávače mohou používat vyhledávání řetězce na webech a poskytovat uživateli přesnější výsledky, které obsahují shodu s vyhledávaným textem.

Druhou oblastí je úprava obrazu a video filtry. Video je pouze rychlý sled více obrázků za čas. Obraz může být upraven za účelem uměleckým nebo za účelem připravit obraz k dalšímu zpracování. Mezi operace uměleckých filtrů patří vyhlazování obrazu, redukce barevného prostoru (například sépiový nebo černobílý filtr), odstraňování šumu v obraze, rozmazání obrazu nebo detekce hran (například pro vytvoření obrysu z fotky pro rytinu do dřeva nebo na sklo). Účelem uměleckých filtrů je ohromit a zaujmout pozorovatele, využití je v reklamních a prezentačních aplikacích. Předzpracování obrazu znamená upravit obraz do vhodné reprezentace, kterou dokáže využít navazující aplikace. Může se jednat o zjištění obrysu objektu v obraze, podle kterého navazující aplikace v podobě klasifikátoru objekt zařadí do určité kategorie a vykoná odpovídající akci. Příkladem je klasifikace projíždějících aut za účelem analýzy nebo klasifikace výrobků projíždějících na pásu za účelem určení jejich způsobu dopravy.

Třetím okruhem je akcelerace filtru audio signálu. Audio signál je využívám v aplikacích s rozpoznáváním hlasu (příklad konkrétní aplikace je hlasový asistent v chytré domácnosti nebo v mobilu) a před zpracováním zvuku je potřeba odfiltrovat okolní zvuky, které nejsou spojeny s řečí. Podobným principem je rozpoznávání slov z řeči nebo odlišení více řečníků v diskuzi, případně překlad z mluveného slova. Všude v těchto případech je potřeba odfiltrovat nepotřebné zvuky z okolí. Dalším využitím filtrů audio signálu je v hudebním průmyslu. Používají se filtry určitých frekvencí (basy, středy, výšky) a potlačování ostatních pro dosažení chtěného efektu. Při mixování a úpravě hudby jsou používány různé efekty zpomalování, zrychlování, ozvěn a řada dalších, které jsou realizovány taktéž pomocí filtrů.

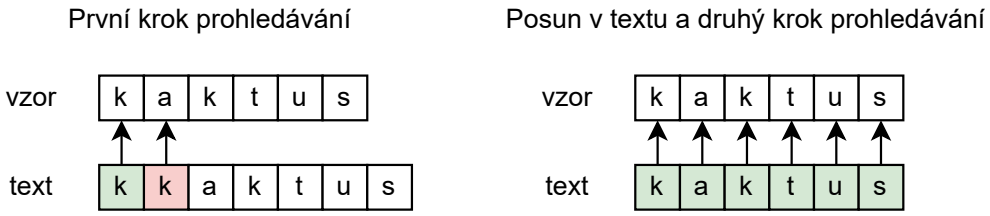
Poslední vybrané využití hardware akcelerace je z oblasti internetové bezpečnosti. Komunikace z lokální sítě bývá oddělena bránou (gateway), přes kterou proudí data z veškerých počítačů v lokální síti. Z důvodu bezpečnosti je vhodné komunikaci do internetu skrze tuto bránu vymezit na důvěryhodné servery a aplikace. Ostatní nežádoucí pokusy o navázání komunikace je potřeba co nejefektivněji odfiltrovat, aby zbytečně nebrzdily užitečnou komunikaci. Společnosti, jejichž podnikatelským záměrem jsou internetové služby (obchody, cloudy a spousta dalších), musí čelit DoS (Denial of Services) útokům, které se snaží danou službu vyřadit z provozu zasíláním nadměrného množství požadavků. Tyto požadavky na straně serveru nestíhají být zpracovány a ostatním uživatelům brání k využívání služby. Vhodným filtrem, který lze implementovat v hardware, se dají tyto požadavky filtrovat a hlavní služba se jimi nemusí ani zabývat, protože se k ní nedostanou.

Následující sekce řeší jednu problematiku z výše uvedených oblastí a představují vybrané algoritmy, kterými je možno problémy řešit. První sekce 3.1 řeší vyhledávání v textu, v sekci 3.2 jsou obrazové filtry, v sekci 3.3 jde o filtrování audio signálu, v poslední sekci 3.4 jde filtrování internetových paketů.

3.1 Vyhledávání řetězce v textu

Prvním představeným algoritmem je vyhledávání podřetězce hrubou silou. Metoda prochází postupně znak po znaku celý text. Vždy při přejití na nový znak se začnou porovnávat následující znaky s vyhledávaným vzorem. Pokud nastane shoda, je signalizován nález, pokud shoda nenastane, algoritmus se posune na další znak a proces porovnávání dalších znaků se vzorem opakuje. [17]

Na obrázku 3.1 je znázorněno vyhledání slova *kaktus* v řetězci *kkaktus*. Algoritmus na obecném procesoru probíhá sekvenčně, znaky se vzorem jsou porovnávány postupně. Tento krok jde v hardware paralelizovat a porovnat znaky se vzorem naráz paraleně. U paralelní varianty může nastat problém, že prohledávaný text je kratší než vzor a je potřeba prohledávaný text doplnit nějakou výplní na velikost vzoru. Nabízí se použít znaky, které se nemohou vyskytovat v prohledávaném textu nebo použít znaky, které se nevyskytují ve vzoru, tudíž nemůže nastat shoda. Druhou možností je provést analýzu konce textu a výplň zvolit ze symbolů, které se sice mohou objevit ve vyhledávaném textu, ale jejich posloupnost s návazností na konec řetězce se nikdy neshodne se vzorem.



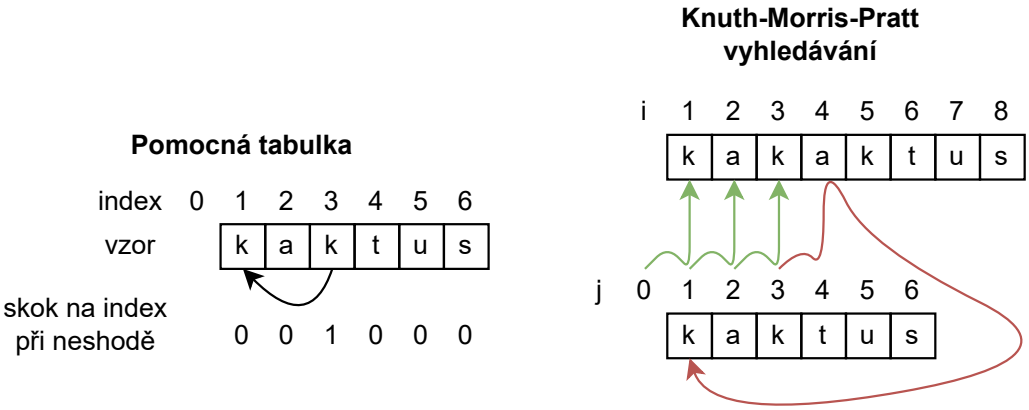
Obrázek 3.1: Vyhledání řetězce v textu hrubou silou.

O něco sofistikovanější je algoritmus Knuth-Morris-Pratt. Přístup je podobný jako v případě hrubé síly, avšak při neshodě se algoritmus nevrací zpět v prohledávaném textu jako v sekvenčním algoritmu hrubou silou. Před samotným prohledáváním musí být vytvořena pomocná tabulka, která má uloženy indexy posunu v rámci vzoru v případě neshody. [17]

Na obrázku 3.2 je příklad vyhledání slova *kaktus* v řetězci *kakaktus*. Z vyhledávaného vzoru *kaktus* se nejdříve vytvoří seznam prefixů.

- seznam prefixů: k, ka, kak, kakt, kaktu, kaktus

Pro vytvoření tabulky se hledá nejdelší část dále v řetězci, která je zároveň prefixem. První k je prefixem sebe samého. Následující a není prefixem slova *kaktus*. Následuje druhé k, které je prefixem *kaktus*, poznačí se index skoku na první k. Za druhým k následuje t a kt ani t není prefixem slova kaktus. Dále se pokračuje obdobně a u, s nejsou prefixem slova *kaktus*.

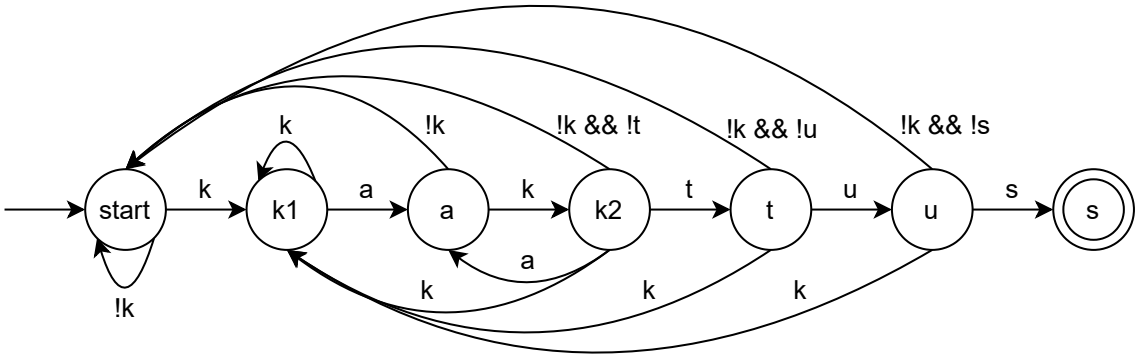


Obrázek 3.2: Algoritmus vyhledání textu Knuth-Morris-Pratt s pomocnou tabulkou indexů ve vyhledávaném vzoru.

Vzor má pomocnou pozici 0, na které zároveň začíná. Vyhledávání začíná na obou indexech $i, j = 0$, vzor se porovnává s indexem $j + 1$:

1. $i[0] = k$, $j[0 + 1] = k$ – shoda, inkrementují se indexy na $i = 1$, $j = 1$
2. $i[1] = a$, $j[1 + 1] = a$ – shoda, inkrementují se indexy na $i = 2$, $j = 2$
3. $i[2] = k$, $j[2 + 1] = k$ – shoda, inkrementují se indexy na $i = 3$, $j = 3$
4. $i[3] = a$, $j[3 + 1] = t$ – neshoda, vznikla neshoda na indexu $j = 3$, v pomocné tabulce je na indexu 3 poznačeno 1, tj. ukazatel se nastaví $j = 1$
5. $i[3] = a$, $j[1 + 1] = a$ – shoda, inkrementují se indexy na $i = 4$, $j = 2$
6. stejným principem probíhá porovnávání a zvyšují se indexy, dále nastává již samá shoda

Vyhodnocování lze převést na stavy a vytvořit konečný stavový automat jako na obrázku 3.3, který je implementovatelný na FPGA. Konečné automaty se vytvářejí systematickým a deterministickým postupem, který usnadňuje analýzu toku programu a ladění chyb. Také jsou vhodným prostředkem pro převod do jiných reprezentací a existuje pro ně řada optimalizačních algoritmů a vizualizačních nástrojů.



Obrázek 3.3: Algoritmus Knuth-Morris-Pratt převedený na konečný automat a vyhledávající výskyt slova kaktus.

3.2 Obrazový filtr

Obrazový filtr pracuje s dvourozměrným prostorem dat. Obraz je složen z pixelů, přičemž každý pixel má 3 barevné složky – červenou (R), zelenou (G), modrou (B). Každá barevná složka je většinou určena 8 nebo 16 bitovou hodnotou. Operace nad obrazem mohou pracovat nad samotnými body izolovaně nebo brát v úvahu i okolní pixely. Základní operací pracující s každým bodem odděleně je například převod barevného obrazu na stupně šedi. Převod je založen na informaci, s jakou člověk vnímá jednotlivé barevné složky:

- $I = 0,299 \cdot R + 0,587 \cdot G + 0.114 \cdot B$ [16]

Tato metoda bude rozšířena o zachování červené barvy za účelem vytvořit umělecký filtr. Podmínka pro zachování původní červené barvy je v této práci vytyčena několika rovnicemi, které vycházejí z barevného modelu HLS. Hlavními složkami tohoto barevného modelu je odstín (hue), svítivost (luminosity) a sytost (saturation) [11].

Odstín barvy vychází z vlnové délky viditelného světla, které je v rozmezí 350 nm – 750 nm. Výčet všech odstínů barev v horizontálním směru se nazývá barevné spektrum a lze ho vidět na obrázku 3.4. Napravo od spektra se nachází běžnější počítačová reprezentace v podobě kruhu, ve kterém je naznačen ostrý úhel mezi 15° a 345°. Odstíny uvnitř úhlu se berou jako červená barva k zachování, ostatní odstíny budou převedeny na odstíny šedi. Tento i následující parametry jsou nastaveny čistě dle subjektivního dojmu. Vnímání barev jednotlivými lidmi se může lišit a jde pouze o individuální dojem. Definovat konkrétní hranici, co je ještě červená barva a pokud se změní jedna hodnota z R,G,B složek, tak se již nejdená o červenou, je obtížný úkol, který není předmětem této práce.

Obrázek 3.4: Odstíny barev vyjádřeny viditelným spektrem.

Výpočet odstínu z R, G, B složek se rozpadá na 6 rovnic podle toho, ve kterém kvadrantu se odstín nachází v kruhu. První rovnice 3.1 navíc připouští možnost, že dělitel je roven 0.

Dělení nulou je nedefinovaná operace a tuto podmínku je nutné explicitně ošetřit a položit odstín roven 0 místo počítání rovnice, jinak by došlo k výjimce.

$$R \geq G \geq B \quad | \quad H = 60 \cdot \left(\frac{G - B}{R - B} \right) \tag{3.1}$$

$$G > R \geq B \quad | \quad H = 60 \cdot \left(2 - \frac{R - B}{G - B} \right) \tag{3.2}$$

$$G \geq B > R \quad | \quad H = 60 \cdot \left(2 + \frac{B - R}{G - R} \right) \tag{3.3}$$

$$B > G > R \quad | \quad H = 60 \cdot \left(4 - \frac{G - R}{B - R} \right) \tag{3.4}$$

$$B > R \geq G \quad | \quad H = 60 \cdot \left(4 + \frac{R - G}{B - G} \right) \tag{3.5}$$

$$R \geq B > G \quad | \quad H = 60 \cdot \left(6 - \frac{B - G}{R - G} \right) \tag{3.6}$$

Svítivost (jas) vyjadřuje intenzitu energie ze zdroje viditelného světla. Hodnota je v intervalu $\langle 0, 1 \rangle$ a říká, jak moc je daný odstín barvy světlý, jak je barva jasná. Model jasu s intervalem tmavosti a světlosti barvy je zachycen na obrázku 3.5.

Obrázek 3.5: Interval jasu (množství energie) v odstínu barvy.

Rovnice jasu 3.7 pracuje s normalizovanými hodnotami barev $R_n = \frac{R}{255}$, $G_n = \frac{G}{255}$, $B_n = \frac{B}{255}$. Jako akceptovatelný interval jasu byl zvolen rozsah $\langle 0.2, 1 \rangle$.

$$L = \frac{1}{2} \cdot (max(R_n, G_n, B_n) + min(R_n, G_n, B_n)) \tag{3.7}$$

Sytost barvy je hodnota z intervalu $\langle 0, 1 \rangle$. Sytá barva je krásně živá a postupně s ubývající sytostí bledne až do šedotónového odstínu. Vliv sytosti na barvu lze vidět na obrázku 3.6.

Obrázek 3.6: Interval vlivu sytosti na barvu.

Rovnice pro výpočet sytosti 3.8 používá ve výpočtu opět normalizované hodnoty barev a hdonotu jasu. V jednom případě vyjde dělitel roven 0 a místo výpočtu rovnice se musí přiřadit hodnota systosti. Pro filtr v této práci byl nastaven práh sytosti červené barvy na

interval $\langle 0.4, 1\rangle$.

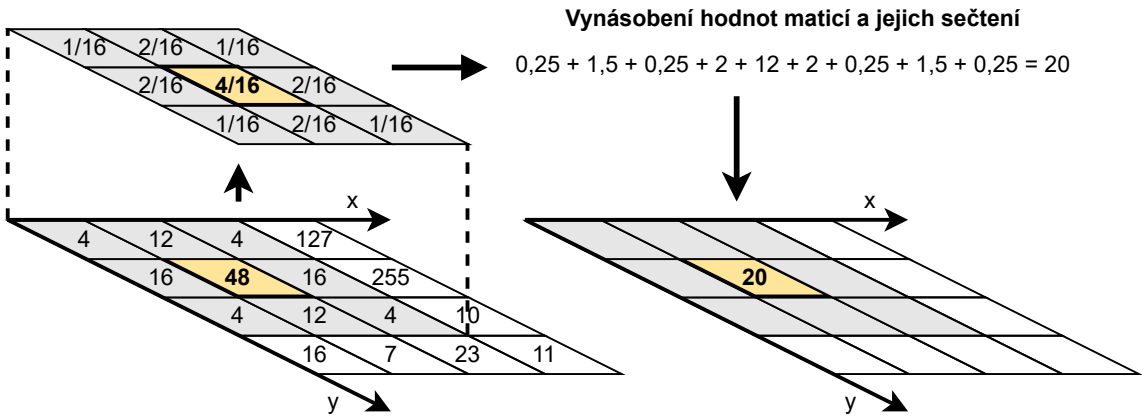
$$L < 1 \quad | \quad S = \frac{max(R_n, G_n, B_n) - min(R_n, G_n, B_n)}{1 - |2L - 1|} \tag{3.8}$$

$$L = 1 \quad | \quad S = 0 \tag{3.9}$$

Po provedení barevného převodu může obraz obsahovat skokové změny barvy (rychlé frekvence), které lze vyhladit dolnopropustním filtrem (Gaussův filter). Tento filtr již pracuje i s hodnotami okolních pixelů. Maska filtru má tvar matice, kterou lze vidět na obrázku 3.7. Tato matice se postupně přikládá na každý pixel v původním obraze a aplikuje na něj funkci danou použitým filtrem. Vyhlazení je potřeba provést pro každý barevný kanál R, G, B zvlášť. Matice filtru zasahuje do sousedních pixelů, ve své funkci zohledňuje chování okolí každého bodu. Speciální případ tvoří krajní body obrazu, kterým chybí někteří sousedi.

Řešení krajních bodů mohou být různá. Krajní body se mohou vynechat a matice filtru (maska) se musí upravit na počítání pouze s dostupnými body. Okraje si lze domyslet a počítat, že jsou bílé nebo černé, případně průměrné hodnoty. Další možností je rotovat obraz a v místech, kde končí levý okraj, přenést začátek pravého okraje a podobně v místech, kde končí dolní část obrazu, přenést horní část obrazu.

Gaussův vyhlazovací filtr z obrázku 3.7 vynásobí pixely v dosahu jeho masky s hodnotou v matici. Tyto hodnoty sečte a získá hodnotu pixelu v novém obrazu. Toto násobení provádí pro každý barevný kanál RGB.



Obrázek 3.7: Vyhlazovací maska a příklad vyhlazení Gaussovým filtrem.

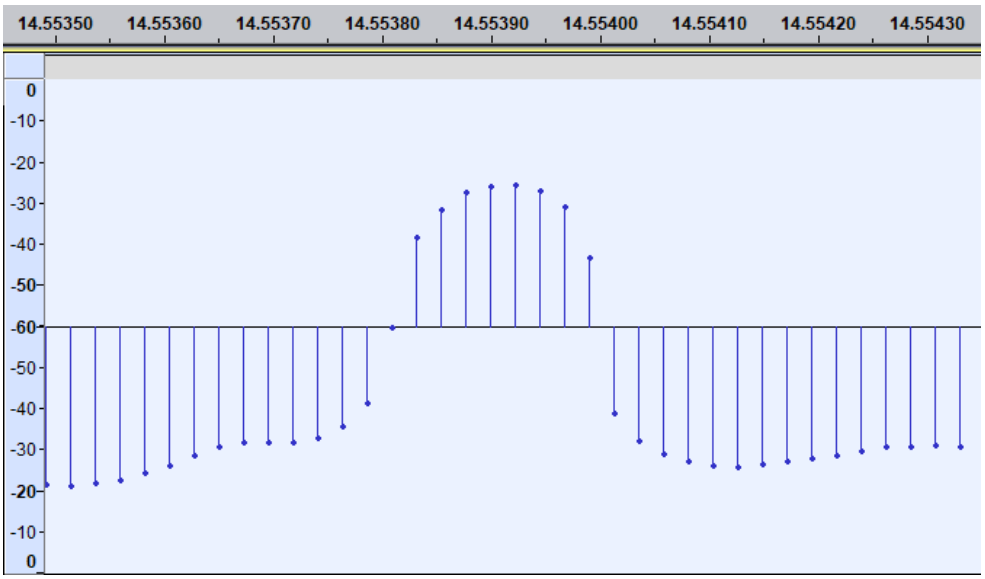
Kromě převodu na černobílý obraz a vyhlazení obrazu existuje spousta dalších filtrů, které se liší velikostí masky (matice) a svými koeficienty. Například ostřící filtr může posloužit k zaostření vyfotografovaných poznávacích značek jedoucích aut. Změnou intenzity obrazu lze upravit příliš tmavé obrazy lišící se drobnými odstíny tmavých barev (například ve vesmíru) a detekovat dříve nerozpoznané objekty. Podobným způsobem lze upravit příliš světlé obrazy (například snímky ovlivněné slunečním svitem) a opět lépe rozpoznat objekty na snímku. Dále lze vyrobit filtr, který převádí 16 bitové hodnoty pixelů na 8 bitové a docílit tak kompresi dat. Dalším příkladem je převod černobílého obrazu na pouze bílé a černé pixely. Na takových snímcích lze jednoduše rozpoznat obrys objektu, čehož se využívá při rozpoznávání a klasifikaci.

3.3 Zvukový filtr

Informace o digitálních signálech pochází z literatury [2]. Zvuková data mají podobu jednorozměrného vektoru, která lze vynést do grafu. Obrázek 3.8 zachycuje krátký úsek nahrávky s konkrétním navzorkovaným signálem. Vzorky odpovídají tlaku vlny, která působila na nahrávací mikrofon. Vzorky jsou odebírány konstantně po stejných časových úsecích. Horizontální osa je v sekundách, vertikální osa je v decibelech. Decibel je bezrozměrná jednotka, udává pouze poměr dvou čísel. U digitálního zvuku jde o poměr aktuální úrovně napětí ku referenčnímu napětí audio techniky. Hodnota $-60\,\mathrm{dB}$ vyjadřuje nejmenší hlasitost slyšitelnou člověkem a hodnota $0\,\mathrm{dB}$ je maximální hlasitost, kterou je audio technika schopná přehrát bez zkreslení, cokoliv nad $0\,\mathrm{dB}$ je omezeno výkonem reproduktoru. Hodnoty dB jsou udávány v logaritmické doméně. Rovnice 3.10 ukazuje příklad výpočtu, kdy je poměr aktuálního a referenčího napětí stejný a dosáhlo se maximální hodnoty $0\,\mathrm{dB}$.

$$\log\left(\frac{u}{u_{ref}}\right) = \log(1) = 0\,dB \tag{3.10}$$

Druhý pohled na zvukový signál je ve frekvenční doméně. Člověk od přírody vnímá zvuk (hlasitost i frekvenci) v logaritmickém měřítku. Rozdíl frekvencí $440\,\mathrm{Hz}$ a $880\,\mathrm{Hz}$ připadá člověku podobný jako rozdíl $880\,\mathrm{Hz}$ a $1760\,\mathrm{Hz}$. Frekvence je dána počtem vzorků za jednotku času, například audio nahrávka dosahuje 44100 vzorků za sekundu, tj. vozrkovací frekvence je $f_s = 44100\,\mathrm{Hz}$. Maximální reprezentovatelná frekvence je omezena podle Nyquistova teorému na polovinu vzorkovací frekvence, tj. pouze $22050\,\mathrm{Hz}$.



Obrázek 3.8: Úryvek zvukové nahrávky.

Jedním z možných typů filtrů jsou filtry s konečnou impulzní odezvou známé pod zkratkou FIR. Impulzní odezva je měřena na výstupu filtru, když je do něj přiveden jednotkový pulz. Filtrům s konečnou odezvou se musí hodnota na výstupu po přivedení jednotkového pulzu ustálit na nějaké hodnotě v konečném čase. Základní rovnice filtru má tvar 3.11, ve které je potřeba stanovit počet členů a váhové koeficienty h.

$$y[n] = h[k]x[n-k] + h[k]x[n-k] + h[k]x[n-x] + \ldots \tag{3.11}$$

Koeficient $k = 1, 2, 3 \ldots$ se v rovnici postupně zvyšuje a odpovídá počtu členům filtru. Počet členů filtr lze vypočítat Harrisovou aproximací 3.12.

$$N \approx \frac{A \cdot f_s}{22 \cdot BW} \tag{3.12}$$

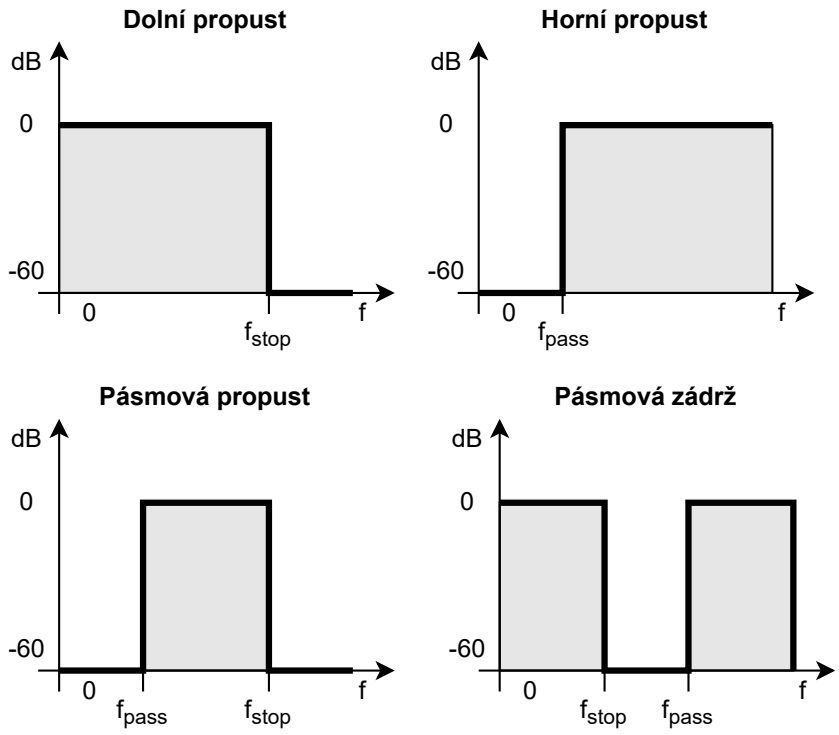
- A je počet decibelů, s jakým dochází k poklesu amplitud frekvencí
- f_s je vzorkovací frekvence signálu
- BW je pásmo $|f_{pass} - f_{stop}|$, ve kterém dochází k poklesu

Frekvence v slyšitelném spektru člověka jsou děleny na nízké, střední a vysoké s následujícími frekvenčními rozsahy:

- nízké – 20 Hz až 300 Hz
- střední – 300 Hz až 4 kHz
- vysoké – 4 kHz až 20 kHz

Filtry lze rozdělit do skupin podle toho, zda využívají informaci z okolí. První skupinou jsou filtry, které pracují pouze s jedním vzorkem. Může se jednat o úpravu hlasitosti (amplitudy) a změna jednoho vzorku nevyžaduje žádnou informaci z okolních vzorků.

Druhou skupinou jsou filtry závislé na okolí signálu. Jde o zpožďovací efekty – hodnota signálu se promítne s určitou váhou do pozdějšího vzorku. Nebo se jedná o frekvenční filtry, které vhodnou kombinací okolních prvků upravují hlasitosti signálů na daných frekvencích. Tato práce se zabývá návrhem frekvenčního filtru, který propouští nízkofrekvenční signál beze změny a hlasitost vyšších frekvencí tlumí – tzv. dolnopropustní filtr. Základní typy filtrů a jejich ideální utlumovací charakteristiky lze vidět na obrázku 3.9.



Obrázek 3.9: Frekvenční charakteristiky základních audio filtrů.

3.4 Filtrace internetových paketů

K filtrování paketů slouží aplikace označována jako brána firewall. Filtrování probíhá na základě seznamu pravidel. Na seznam pravidel existují dva pohledy. Jeden pohled je, že pravidla v seznamu jsou povolovací a co není explicitně povoleno v seznamu pravidel, tak je zakázáno. Propouští se pouze pakety, které splňují pravidla uvedená v seznamu. Druhý pohled je, že pravidla v seznamu jsou zakazovací. Pokud je pravidlo uvedeno v seznamu a příchozí paket pravidlo splní, tak se na paket pohlíží jako na nežádoucí paket a cílem je ho odfiltrovat. Ostatní pakety se propouští.

Pravidla sloužící k filtraci paketů lze stanovovat na základě Internetového Protokolu verze 4 (IPv4), který je jedním z používaných protokolů na přenos dat v internetu. Formát hlavičky a význam IPv4 protokolu specifikuje standard RFC791 [6]. Kladením požadavků na položky a hodnoty v IPv4 protokolu lze utvářet pravidla, na základě kterých probíhá třídění paketů. Filtrování nežádoucích paketů se používá za účelem dosažení bezpečnosti komunikace a umožnit přenášet data pouze mezi ověřenými zdroji nebo zabránit útokům, které se snaží zahltit síť posíláním nadměrného množství paketů. Struktura hlavičky IPv4 a její položky jsou na obrázku 3.10.

Obrázek 3.10: Struktura hlavičky IPv4 paketu dle specifikace RFC791. Velikosti položek jsou zobrazeny v bitech.

Popis položek IPv4 hlavičky z obrázku 3.10:

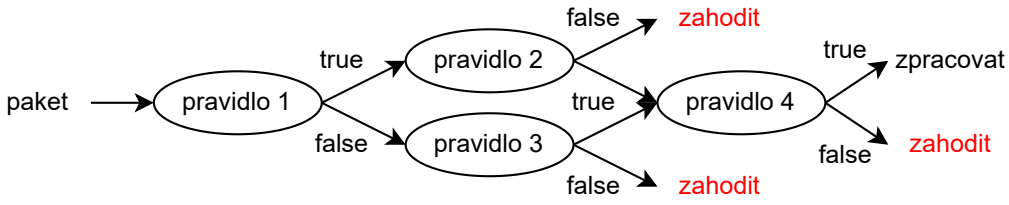
- Verze – verze protokolu určuje podobu hlavičky paketu.
- Velikost hlavičky – celková velikost hlavičky udávaná v počtu 32 bitových bloků.
- Typ služby – využívá se v sítích, které akceptují prioritní a méně důležité pakety. Při zahlcení sítě a přeposílání paketů se zahazují méně důležité pakety.
- Celková velikost – celková velikost paketu (hlavička i data) udávaná v počtu osmibitových bloků.
- Identifikace – identifikační číslo zadané odesílatelem.
- Příznaky – první bit je rezervovaný. Druhý bit určuje, zda je možné paket po cestě fragmentovat (rozdělit na více menších paketů). Třetí bit nese informaci o tom, jestli je aktuální paket poslední, nebo následuje více paketů (došlo k fragmentování původního paketu na více částí, které je potřeba na konci opět složit).

- Offset fragmentu – používá se při fragmentaci paketů, udává pozici dat v původním nefragmentovaném paketu. Pozice dat je v blocích po 64 bitech.
- Doba života – při každém přeposlání paketu na další router se doba života snižuje o 1. Pokud dosáhne hodnoty 0, pak je paket zahozen. Předchází se tak zacyklení přeposílání paketů.
- Protokol – protokol na další úrovni, kterou zapouzdřuje IP protokol.
- Kontrolní součet hlavičky – hlavička se rozdělí na 16 bitové bloky. Všechny tyto bloky jsou sečteny (kromě checksum pole). Přetečený vyšší řád je přičten k 16 bitové hodnotě součtu. Vytvoří se jedničkový doplněk tohoto součtu.
- Zdrojová adresa – IPv4 adresa odesílatele.
- Cílová adresa – IPv4 adresa příjemce.
- Další volby – další volby hlavičky, které nejsou povinné a nemusí se použít.
- Zarovnání – při použití nepovinných voleb se na konec vkládá výplň k zarovnání na velikost 32 bitových bloků.

Pravidla jsou ve formě logických `AND` a `OR` výrazů. Příkladem konkrétního pravidla, které omezuje komunikaci na dvě cílové adresy z jedné zrojové adresy a pouze protokol TCP. Tyto údaje jsou bitové informace, které lze paralelně velmi rychle a efektivně porovnávat.

- `src=192.0.128.1 AND (dst=192.0.60.1 OR dst=192.0.60.2) AND proto=TCP`

Jedním možným přístupem je rozhodovací strom, jehož základní myšlenka je na obrázku 3.11. Strom sestává z ulzů pravidel. Paket je postupně prosévám přes pravidla, která rozhodují o jeho dalším posunu. Buď paket pravidlu vyhoví a je posunut ke zkoumání další vlastnosti nebo je paket označen k zahození. Pokud paket projde stromem až do úspěšného cílového uzlu, je označen za vyhovující a může být zpracován procesorem.



Obrázek 3.11: Filtrace paketů rozhodovacím stromem.

Kapitola 4

Návrh implementace ukázkových aplikací

Platforma Pynq Z2 je založena na operačním systému Linux. Programovat lze ve všech programovacích jazycích, ke kterým je na Linuxu dostupný překladač. Nicméně preferovaným jazykem je Python, u kterého jsou v základu dostupné vývojové prostředí i podpůrné knihovny pro práci s FPGA čipem.

Vývojovým prostředním je Jupyter Notebooks interaktivní dokument. Dokument tvoří vstupní pole jednoho ze dvou typů – Markdown nebo kódové pole. V Markdown poli lze sázet formátovaný text, vkládat rovnice, obrázky, webové odkazy. Kódové pole se chová jako interpretr jazyka Python a provádí kód. Kódové úseky a Markdown popisy se dají libovolně kombinovat a prokládat.

Z Pythonu se volají knihovní funkce zastřešující přístup ke stahování a nahrávání bitstream konfigurace do FPGA. Vlastní návrh hardware obvodu se implementuje v aplikaci Vitis HLS. Popis hardware probíhá na vyšší úrovni abstrakce v jayzce C. Následně se z jazyka C generuje VHDL popis použitelný jako IP komponenta. Nástroj Vivado umožňuje importovat nově vytvořené IP bloky a použít je v blokovém návrhu.

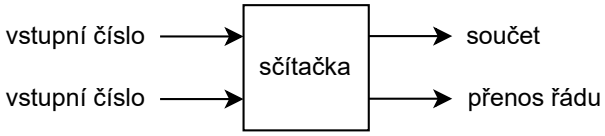
První navrhovanou aplikací je jednoduchá sčítačka pro součet dvou čísel a zachycení přenosu přetečení na vyšší řád (sekce 4.1). Druhý návrh aplikace je vyhledávání v textu (sekce 4.2). Třetí návrh aplikace řeší strukturu a zpracování obrazových dat (sekce 4.3). Čtvrtá aplikace se zabývá reprezentací zvukového signálu a jeho úpravy (sekce 4.4). Posledním návrhem je filtr internetových paketů (sekce 4.5).

Každá navržená aplikace bude mít svůj Jupyter Notebook dokument s doprovodným obsahem v podobě textu, obrázků, rovnic a případně dalšího materiálu, který danou problematiku doplní. Zároveň ke každé aplikaci bude dostupná vlastní hardware knihovna, nazývaná Overlay. Overlay poskytuje metody zapouzdřující přístup k implementaci funkce v hardware. Oddělení jednotlivých implementací do různých knihoven zjednoduší pohled na blokové schéma hardware implementace části aplikace. Blokové schéma bude obsahovat pouze prvky týkající se konkrétní problematiky a bude snáž upravitelné a rozšiřitelné.

Strukturu aplikací tvoří dvě hlavní části. První z nich je softwarová implementace s cílem vhodně připravit data ke zpracování na FPGA a přebírat výstupy zpracovaných dat z FPGA k jejich dalšímu využití nebo zobrazení uživateli. Druhá část se věnuje realizaci algoritmů na FPGA a poskytnutí vhodného komunikačního rozhraní mezi doménami hardware a software.

4.1 Návrh jednoduché sčítačky

Blok sčítačky je zachycen na obrázku 4.1. Vstupem jsou dvě 32 bitová čísla, výstupem sčítačky je 32 bitový součet a 32 bitová hodnota indikující přetečení do vyššího řádu. K této indikaci stačí pouze jednobitová hodnota, ale interně jsou přenosy v hardware zarovnány na na 32 bitů.



Obrázek 4.1: Sčítačka dvou 32 bitových čísel.

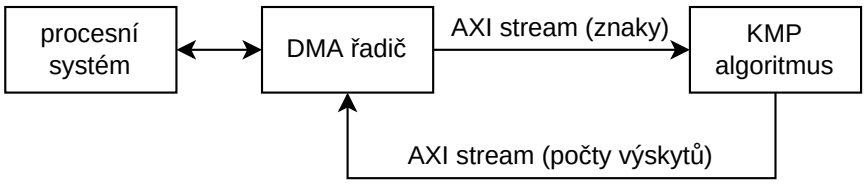
Úkolem softwarové části je připravit dvě čísla a poslat je do sčítačky implementované v FPGA. Komunikace v tomto případě není nijak kritická a jako komunikační rozhraní mezi procesním systémem a FPGA může být použito AXI GP (General Purpose). Přes stejné rozhraní je poskytnut výsledek, který si převezme softwarová část.

Cílem této aplikace je prezentovat kompletní postup vytvoření aplikace od hardware návrhu obvodu v nástroji Vitis HLS, provedením jeho syntézy do jazyka VHDL a vygenerování IP bloku použitelného v nástroji Vivado, kde se bloky napojují na procesní systém.

4.2 Návrh vyhledání řetězce v textu

Softwarová část se stará o přípravu dat, načtení textových souborů, ve kterých se bude provádět vyhledávání. U prohledávání velkého množství souborů nebo obsáhlých souborů již jde o časově náročnou operaci a komunikaci. Zpracování souborů je vhodné řešit skrze DMA řadič, který je dostupný v podobě IP (Intellectual Property) komponenty. Z DMA řadiče je vyvedeno AXI-stream rozhraní popsané v kapitole 2.1, z kterého proudí znak po znaku.

Zjednodušené schéma zapojení spolu s rozhraním akcelerátoru je na obrázku 4.2. Konec jednoho proudového přenosu (přenosu souboru) je AXI rozhraním indikován signálem *TLAST*. Na výstupu akcelerátoru je opět AXI-stream rozhraní s polem hodnot výskytů v jednotlivých souborech.



Obrázek 4.2: Blokové schéma HW akcelerátoru vyhledávání v textu.

Vnitřní část vyhledávače vychází z algoritmu Knuth-Morris-Pratt (dále jen KMP) představeného v sekci 3.1. Algoritmus je založen na konečném automatu. Demonstrační aplikace bude vyhledávat některá anglická slova (*bear*, *duck*, *shark*) a počítat jejich výskyty, tj. výstupem aplikace bude pole s hodnotami výskytů jednotlivých slov. Výskyty nalezených slov lze použít například k doporučování literatury. Nebo k opačnému účelu, vyhledávaná slova

zaměnit za nevhodná slova a podle jejich výskytů skrývat literární díla, webové stránky nebo komentáře osobám s nepotvrzeným věkem.

Popis takového konečného automatu, i když docela jednoduchého, vede na komplikovaný graf, který není přehledný. Proto jsou jednotlivé stavy a přechody znázorněny přechodovou tabulkou 4.1. Řádek tabulky jsou stavy konečného automatu, sloupce jsou symboly právě na vstupu a uprostřed buněk jsou následující stavy, do kterých se automat přesune. V tabulce se nachází jeden speciální symbol a jeden speciální stav. Symbol hvězdičky (*) nese význam všech ostatních neuvedených symbolů, které se ale mohou objevit na vstupu automatu. Stav označený mřížkou (#) má význam počátečního stavu, ve kterém konečný automat začíná.

stavy	**vstupní symboly**										
	a	**b**	**c**	**d**	**e**	**h**	**k**	**r**	**s**	**u**	*****
#	#	b	#	d	#	#	#	#	s	#	#
b	#	b	#	d	be	#	#	#	s	#	#
be	bea	b	#	d	#	#	#	#	s	#	#
bea	#	b	#	d	#	#	#	bear	s	#	#
bear	#	b	#	d	#	#	#	#	s	#	#
d	#	b	#	d	#	#	#	#	s	du	#
du	#	b	duc	d	#	#	#	#	s	#	#
duc	#	b	#	d	#	#	duck	#	s	#	#
duck	#	b	#	d	#	#	#	#	s	#	#
s	#	b	#	d	#	sh	#	#	s	#	#
sh	sha	b	#	d	#	#	#	#	s	#	#
sha	#	b	#	d	#	#	#	shar	s	#	#
shar	#	b	#	d	#	#	shark	#	s	#	#
shark	#	b	#	d	#	#	#	#	s	#	#

Tabulka 4.1: Přechodová funkce konečného automatu na vyhledávání slov v textu.

4.3 Návrh obrazového filtru

Procesní systém v platformě Pynq Z2 nemá vlastní řadič HDMI, ten je implementován spolu s dalšími řadiči k periferiím v základní knihovně Base Overlay dostupné na oficiální GitHub stránce platformy Pynq [13].

Tento řadič pokrývá vstupní a výstupní operace a pracuje s obrazovými rámci. Poskytuje barevnou konverzi na 8, 16, 24, 32 bitovou barevnou hloubku a pracuje s barevným pořadím pixelu modrá, zelená, červená (BGR). [22]

Umělecký filtr popsaný v kapitole 3.2 je složen ze dvou částí. První filtr řeší převod barevného prostoru. Obraz převádí na černobílý s výjimkou červené barvy. Nejedná se tak o běžný černobílý obraz s jedním barevným kanálem, ale zachovány jsou všechny 3 barevné složky BGR. Převod je prováděn individuálně nad každým pixelem a výsledek není nijak ovlivněn okolními pixely.

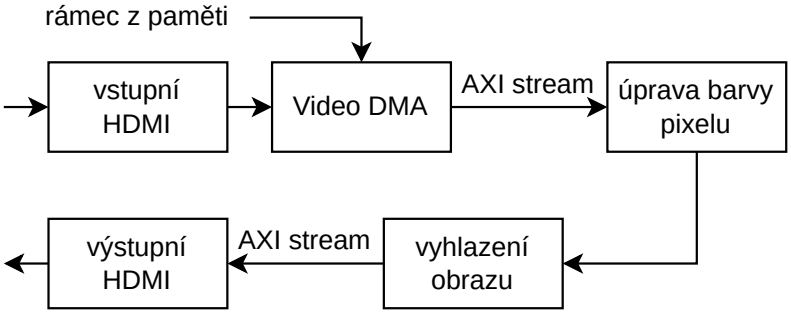
Druhá část filtru se stará o dodatečné vyhlazení obrazu. Filtr je tvořen konvoluční maskou (maticí), která pracuje i s okolím pixelu. Problémovou část tvoří pixely na stranách obrazu a rohové pixely, protože nemají některé sousední pixely. Jednou z možností je provést výplň a obraz předchystat tak, aby měl z každé strany o jednu řadu pixelů víc. Vzniká problém jakou barvu použít jako výplň, nabízí se černá, bílá nebo průměrná barva z okolí. Druhým řešením je pixely po stranách vynechat a začít reálně vyhlazovat až od vnitřních

pixelů obrazu. Tato druhá varianta se jeví jako rychlejší a šetrnější na výpočetní prostředky. Vyhlazení se musí provést nad všemi barevnými kanály BGR zvlášť.

- Gaussův vyhlazovací filtr $\begin{bmatrix} \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \\ \frac{2}{12} & \frac{4}{12} & \frac{2}{12} \\ \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \end{bmatrix}$

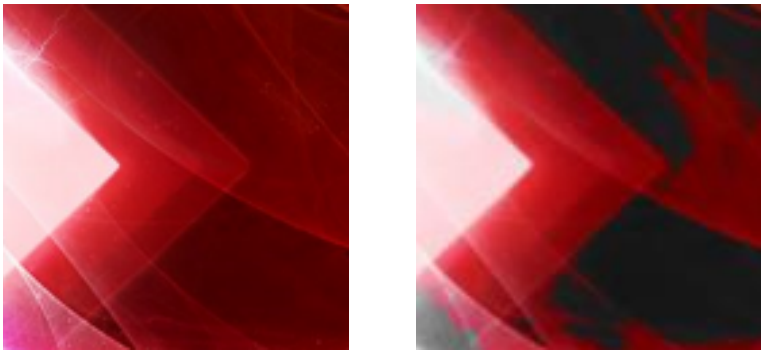
Sled akcí zpracování obrazu se nachází na obrázku 4.3. Vstupní obrazové rámce videa je možné získávat jedním ze dvou vstupů. Prvním způsobem je vstupní HDMI rozhraní, druhým způsobem je použít například knihovnu OpenCV[1] a video uložené v paměti načítat rámec po rámci. Obraz je ukládán v RAM, ze které se přes DMA řadič, s využitím AXI-stream rozhraní, přenáší pixel po pixelu do prvního bloku upravujícího barvu obrazu. Konce přenosů jednotlivých obrazových rámců lze rozeznat signálem *TLAST* v AXI-stream rozhraní.

Po změně barvy se pixel dále přenáší do druhé části filtru, ve které dochází k vyhlazování obrazu. Zde je potřeba načíst a držet uložené kompletní dvě řady pixelů plus 3 další pixely, aby se dosáhlo okolí 3×3, se kterým pracuje vyhlazovací filtr. Takto upravený obraz se odešle do RAM paměti přes DMA řadič a je připraven na vyslání do výstupního HDMI rozhraní. Vstupní i výstupní HDMI rozhraní je dostupné v Base Overlay knihovně [20].



Obrázek 4.3: Návrh filtrace obrazového rámce.

Závěrem je ukázán obrázek 4.4 se dvoumi vzorky o velikosti 128×128 z původního rozlišení 1280×720. Levý vzorek je původní před filtrací a pravý, částečně odbarvený a mírně rozmazaný, je výsledek po filtraci.



Obrázek 4.4: Vzorek z původního a zpracovaného obrazu.

[1] https://pypi.org/project/opencv-python/

4.4 Návrh zvukového filtru

Jedná se o filtr dolní propust, který propouští signál s nízkou frekvencí a vyšší frekvence tlumí. K vytvoření filtru je potřeba stanovit několik parametrů jako propouštěná frekvence, velikost útlumu a rozsah, na kterém dochází k útlumu:

- propouštěná frekvence je stanovena od 0 do 300 Hz
- útlum je nastaven na $A = 24\,\text{dB}$ (0 dB je maximální hlasitost, kterou reproduktor výkonově zvládá a útlum o 60 dB je práh hlasitosti slyšitelné člověkem)
- rozsah frekvencí BW, na kterém bude útlum od 24 dB je nastaven od 300 Hz do 1 kHz
- vzorkovací frekvence zvukového signálu je dána $f_s = 44100\,\text{Hz}$

Dosazením do rovnice 3.12 z kapitoly 3.3 o zvukovém signálu lze získat počet členů filtru, které vyhoví výše zvoleným parametrům.

$$N \approx \frac{A \cdot f_s}{22 \cdot BW} = \frac{24 \cdot 44100}{22 \cdot |300 - 1000|} \approx 69$$

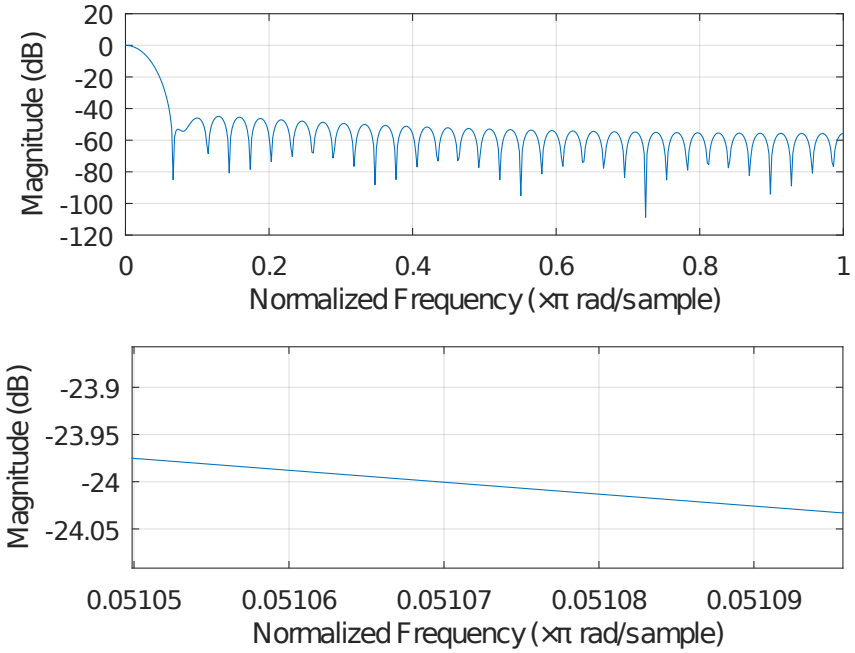
Ze vzorce jsou patrné dvě věci. Čím vyšší je požadovaný útlum, tím více členů filtru je potřeba a čím kratší je frekvenční rozsah útlumu tím je také potřeba více členů. V podobném duchu se nese vzorkovací frekvence, tu je ale snaha většinou neměnit (pokud se nejedná o účely převzorkování nebo komprese). Velký počet členů filtru je příčinou zpoždění výstupu signálu, což je ve zpracování zvuku v reálném čase problém.

Znalost počtu členů lze uplatnit v matematickém software GNU Octave[2] a vygenerovat hodnoty členů filtru.

- vygenerování hodnot členů filtru – `x = fir1(68, 300 / 22050, "low")`, první argument funkce je počet členů filtru bez jednoho, druhý člen je podíl propuštěné frekvence a poloviny vzorkovací frekvence a třetí parametr je typ filtru
- vygenerování frekvenční charakteristiky pro kontrolu – `freqz(x)`

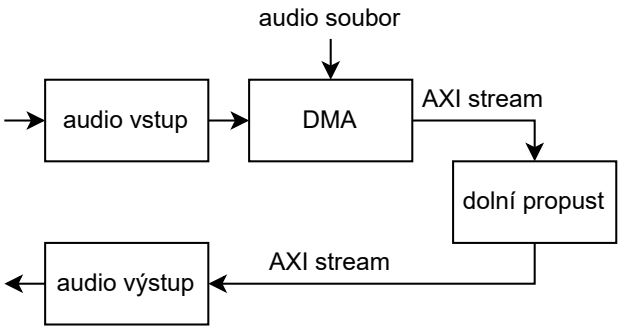
Následující dva grafy 4.5 zachycují frekvenční charakterisitku dolnopropustního filtru. Na vertikální ose je útlum v decibelech (0 je maximální hlasitost ze zařízení a -60 dB je práh slyšitelnosti), horizontální osa je normalizovaná frekvence (0 je 0 Hz a 1 je 22050 Hz). Spodní graf je detailním přiblížením horního grafu v oblasti útlumu -24 dB. Odečtením horizontální frekvence a vynásobením jí polovinou vzorkovací vyjde $0,05107 \cdot 22050 \approx 1126\,\text{Hz}$, což přibližně odpovídá útlumu o 24 dB na hranici frekvence 1 kHz.

[2] https://www.gnu.org/software/octave/index



Obrázek 4.5: Frekvenční charakteristika dolnopropustního filtru.

Vstupní i výstupní rozhraní AD/DA převodníku je dosutpné v základní knihovně Base Overlay. Data jsou přenášena do hlavní paměti RAM. Přístup k datům je umožněn DMA řadičem. Následně jsou jednotlivé vzorky zvukového signálu přenášeny jeden za druhým AXI-stream rozhraním do filtru dolní propusti. Vzorky jsou 24 bitové hodnoty, ale interní AXI sběrnice pracuje s 32 bity, na které jsou vzorky zarovnávány. Při posílání vzorků na výstup dochází k odseknutí horních 8 bitů. Důležité je i upozornit, že filtr má zpoždění dané počtem členů filtru (69), tudíž je potřeba se vypořádat s prvními 69 hodnotami a neposílat je na výstup, aby se výsledná nahrávka nezvětšovala v počtu vzorků.



Obrázek 4.6: Blokové schéma audio filtru.

Výsledkem této části sekce je přehledová tabulka 4.2 s hodnotami všech 69 členů filtru.

člen	hodnota	člen	hodnota	člen	hodnota
1	$1,6520 \cdot 10^{-3}$	24	$2,1612 \cdot 10^{-2}$	47	$2,0461 \cdot 10^{-2}$
2	$1,7268 \cdot 10^{-3}$	25	$2,2706 \cdot 10^{-2}$	48	$1,9265 \cdot 10^{-2}$
3	$1,8864 \cdot 10^{-3}$	26	$2,3733 \cdot 10^{-2}$	49	$1,8036 \cdot 10^{-2}$
4	$2,1344 \cdot 10^{-3}$	27	$2,4682 \cdot 10^{-2}$	50	$1,6785 \cdot 10^{-2}$
5	$2,4734 \cdot 10^{-3}$	28	$2,5544 \cdot 10^{-2}$	51	$1,5524 \cdot 10^{-2}$
6	$2,9047 \cdot 10^{-3}$	29	$2,6310 \cdot 10^{-2}$	52	$1,4266 \cdot 10^{-2}$
7	$3,4286 \cdot 10^{-3}$	30	$2,6972 \cdot 10^{-2}$	53	$1,3021 \cdot 10^{-2}$
8	$4.0444 \cdot 10^{-3}$	31	$2,7523 \cdot 10^{-2}$	54	$1,1800 \cdot 10^{-2}$
9	$4,7500 \cdot 10^{-3}$	32	$2,7958 \cdot 10^{-2}$	55	$1,0615 \cdot 10^{-2}$
10	$5,5423 \cdot 10^{-3}$	33	$2,8271 \cdot 10^{-2}$	56	$9,4757 \cdot 10^{-3}$
11	$6,4170 \cdot 10^{-3}$	34	$2,8461 \cdot 10^{-2}$	57	$8,3907 \cdot 10^{-3}$
12	$7,3686 \cdot 10^{-3}$	35	$2,8524 \cdot 10^{-2}$	58	$7,3686 \cdot 10^{-3}$
13	$8,3907 \cdot 10^{-3}$	36	$2,8461 \cdot 10^{-2}$	59	$6,4170 \cdot 10^{-3}$
14	$9,4757 \cdot 10^{-3}$	37	$2,8271 \cdot 10^{-2}$	60	$5,5423 \cdot 10^{-3}$
15	$1,0615 \cdot 10^{-2}$	38	$2,7958 \cdot 10^{-2}$	61	$4,7500 \cdot 10^{-3}$
16	$1,1800 \cdot 10^{-2}$	39	$2,7523 \cdot 10^{-2}$	62	$4,0444 \cdot 10^{-3}$
17	$1,3021 \cdot 10^{-2}$	40	$2,6972 \cdot 10^{-2}$	63	$3,4286 \cdot 10^{-3}$
18	$1,4266 \cdot 10^{-2}$	41	$2,6310 \cdot 10^{-2}$	64	$2,9047 \cdot 10^{-3}$
19	$1,5524 \cdot 10^{-2}$	42	$2,5544 \cdot 10^{-2}$	65	$2,4734 \cdot 10^{-3}$
20	$1,6785 \cdot 10^{-2}$	43	$2,4682 \cdot 10^{-2}$	66	$2,1344 \cdot 10^{-3}$
21	$1,8036 \cdot 10^{-2}$	44	$2,3733 \cdot 10^{-2}$	67	$1,8864 \cdot 10^{-3}$
22	$1,9265 \cdot 10^{-2}$	45	$2,2706 \cdot 10^{-2}$	68	$1,7268 \cdot 10^{-3}$
23	$2,0461 \cdot 10^{-2}$	46	$2,1612 \cdot 10^{-2}$	69	$1,6520 \cdot 10^{-3}$

Tabulka 4.2: Tabulka koeficientů audio filtru dolní propusti.

Nezbytné je přiblížit podobu WAV souborů, se kterými pracuje audio řadič i samotný akcelerátor. Struktura souboru je pevně daná. Nejdříve se vyskytuje 44 bajtová hlavička, za kteoru následují data samotných vzorků audio signálu. Na konci souboru ještě mohou následovat volitelné položky v podobě metadat (název umělce, název alba, datum vydání, žánr, ...). V hlavičce se vyskytuje seznam následujících položek [15]:

položka	počet bajtů	význam
chunkID	4	Textový řetězec "RIFF".
chunkSize	4	Velikost souboru.
format	4	Řetězec "WAVE".
subChunk1ID	4	Řetězec "fmt", což je první položka z WAV formátu.
subChunk1Size	4	Velikost zbytku fmt části.
audioFormat	2	Hodnota 1 označující PCM (pulse-code modulation).
numChannels	2	Počet kanálů (1 mono, 2 stereo).
sampleRate	2	Vzorkovací frekvence (8000, 16000).
byteRate	4	Počet bajtů za sekundu.
blockAlign	2	Zarovnání bloků (bajty na vzorek × počet kanálů).
bitsPerSample	2	Počet bitů na vzorek (8, 16, 24, 32).
subChunk2ID	4	Řetězec "data", což je druhá položka WAV formátu.
subChunk2Size	4	Počet dat, které zabírají samotné vzorky.

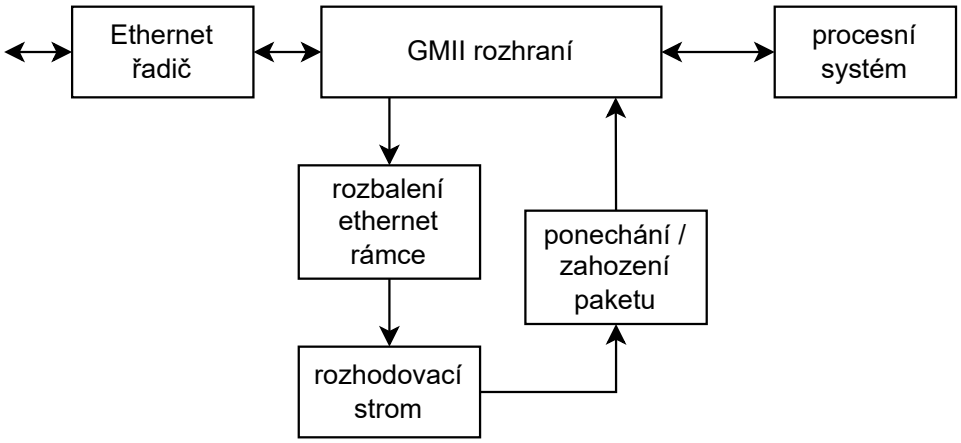
Samotné vzorky jsou při použití více kanálů v souboru proloženy (jeden vzorek z levého, další z pravého dots). Při práci s akcelerátorem je nutné postupovat tak, aby pracoval s audio soubory se dvěmi kanály a 24 bity na vzorek, protože tyto parametry podporuje zvukový řadič. Parametry WAV souboru lze vyčíst z hlavičky nebo s pomocí software. Programem Audacity [3] se dají parametry pohodlně přečíst a jde i převést zvukový soubor do podporovaného formátu k filtraci.

4.5 Návrh filtru internetových paketů

Přenášené internetové pakety mají různou délku, což je problém, protože by v hardware byla potřebná dynamicky se měnící paměť. Nicméně dokument rfc2460 [3] uvádí doporučenou velikost přenášených paketů do 1500 bajtů. S ohledem na tuto informaci lze omezit maximální velikost paketu.

Příchozí paket na ethernetovém řadiči je přes GMII rozhraní převeden na AXI-stream data. Paket je na fyzické vrstvě zapouzdřen do ethernetového rámce [18]. Ethernetový rámec má položky preambule (7 bajtů), oddělovač počátku rámce (1 bajt), cílovou MAC (6 bajtů), zdrojovou MAC (6 bajtů), typ protokolu v rámci (2 bajty), data (46 – 1500 bajtů), kontrolní součet (4 bajty).

Dalším krokem je prosetí paketu rozhodovacím stromem, ve kterém je rozhodnuto o jeho budoucnosti. Navazující blok provede zahození nebo přeposlání paketu do procesního systému. Rovněž je možné z procesního systému pakety i vysílat, ne pouze přijímat.

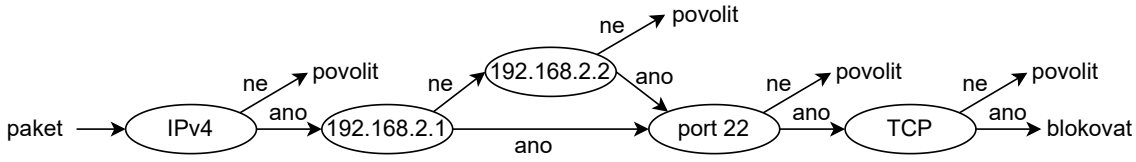


Obrázek 4.7: Návrh klasifikátoru internetových paketů.

Implementace filtru se řídí rozhodovacím stromem popsaným v kapitole 3.4. Pravidla třídící komunikaci jsou založena na omezeních vycházejících z hodnot hlavičky paketu 3.10. Na rozhodovací strom lze pohlížet i jako na konečný automat, ve kterém stavy provádí kontrolu vlastnosti paketu, na základě které je určen přechod do dalšího stavu.

Konkrétní rozhodovací strom založený na následujících prvidlech je uveden na obrázku 4.8. Pakety splňující daná pravidla jsou blokována. Důležité je neblokovat IP adresu a protokoly počítače, ze kterého se programuje Pynq, aby nenastaly problémy s komunikací. Jako konkrétní příklad může být blokování komunikace s následujícími parametry: protokol IPv4, blokované IP adresy 192.168.2.1, 192.168.2.2, přenosový protokol TCP, port 22.

[3] https://www.audacityteam.org/



Obrázek 4.8: Rozhodovací strom na základě zvolených pravidel.

Kapitola 5

Implementace demonstračních aplikací

Implementace jednotlivých aplikací systematicky vychází z návrhů zmíněných v kapitole 4. Implementaci každé aplikace lze rozdělit na 3 dílčí části – implementaci akcelerátoru, provedení blokového propojení a ovládání HW implementace na úrovni kódu v programovacím jazyce Python.

Realizace první části, tedy HW akcelerátoru, se provádí v nástroji Vitis HLS. Tato aplikace je součástí nástrojů Vivado od firmy Xilinx a konkrétně se jedná o verzi 2020.2 s podporou pro platformu Pynq-Z2. Nástroj Vitis HLS vychází z dříve známé aplikace Vivado HLS. Funkce hardware obvodu je popsána vysokoúrovňovým jazykem C++. Syntézou lze vytvořit HDL (angl. Hardware Description Language) popis, ze kterého je možné vygenerovat RTL (angl. Register Transfer Level) popis na úrovni registrů. Společnou vlastností všech vytvářených akcelerátorů je položka v nastavení projektu s výběrem součástky, která je *xc7z020clg400-1*. Tato součástka je pevně vázaná na platformu Pynq-Z2 a je důležitá právě při procesu syntézy, kdy se popis mapuje na technologické prvky (registry, LUT, BRAM...), jejichž dostupnost se na různých platformách liší.

Za zmínku stojí neobvyklá chyba nástroje Vitis HLS, která v době psaní tohoto textu (rok 2022, verze nástroje 2020.2.2) znemožňovala export komponenty jako IP bloku pro aplikaci Vivado. Řešením je nastavení systémového data na rok 2021, se kterým již proběhne vytvoření IP komponenty v pořádku.

Druhým krokem je začlenění navrženého akcelerátoru do procesního systému. Nástroj Vivado obsahuje IP bloky k propojování procesního systému s funkcemi navrženými v programovatelné logice, případně s vlastními navrženými akcelerátory. V této fázi implementace se přidávají vlastní IP návrhy akcelerátorů do repozitáře, který obsahuje souhrn všech IP bloků použitelných v blokovém návrhu. Zároveň se nastavují upravitelné parametry bloků k docílení požadované funkcionality. Napojení komunikace a konfigurace bloků se u jednotlivých akcelerátorů liší v závislosti, s jakými okolními bloky komunikují, jaké mají rozhraní a komunikační protokol.

Overlay knihovna se vytváří z blokového schématu, které ale není zpočátku syntetizovatelné. Nástroj Vivado má v sobě funkci *HDL wrapper*, která návrh připraví do syntetizovatelné podoby. Následuje proces syntézy a implementace, ze které je již možno vytvořit konfigurační soubor bitového toku (známější pod anglickým názvem bitstream) pro FPGA. Tyto 3 kroky představující překlad IP bloku, propojení blokového schématu a vygenerování konfiguračního bitstream souboru pro FPGA jsou automatizovatelné s pomocí

tickle skriptů. Pro každou aplikaci vytvořenou v rámci této práce jsou dostupné následující skripty.

- překlad IP bloků – `build_ip.tcl`
- překlad blokového návrhu – `build_blockdesign.tcl`
- vygenerování konfiguračního souboru pro FPGA – `build_bitstream.tcl`

V Pythonu je poskytován základní ovladač pro přístup k navrženému obvodu v programovatlené logice. Tento ovladač umožňuje zapisovat a číst hodnoty registrů. Tato varianta není příliš vhodná pro složitější akcelerátory, protože by si programátor musel pamatovat velké množství registrů a jejich význam. Vhodné je vytvořit ovladač (driver) a zapouzdřit přístup k bloku do pojmenovaných funkcí s odpovídajícím významem.

5.1 Implementace jednoduché sčítačky

Z pohledu návrhu akcelerátoru v nástroji Vitis HLS jde o malý kus kódu v jazyce C doplněný pomocnými informacemi pro překlad o vstupním a výstupním rozhraní akcelerátoru. První položka *ap_ctrl_none* říká, že není použit žádný specifický protokol komunikace a další 4 položky *s_axilite* označují datové porty připojené na konfigurační AXI sběrnici. Skrze tyto porty bude probíhat komunikace z SW části.

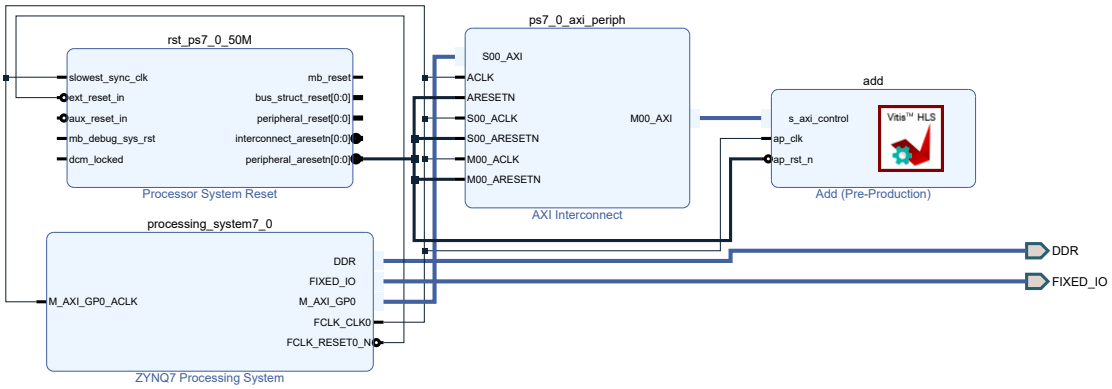
Implementace je složena ze dvou 33 bitových registrů, které jsou vynulovány a jsou do nich uloženy vstupní 32 bitové hodnoty (vstupy a a b). V těchto registrech je provedeno sečtení čísel. Spodních 32 bitů zachycuje součet (výstup c) a horních 32 bitů zachycuje přenos do vyššího řádu (výstup d).

```
void adder(uint32_t a, uint32_t b, uint32_t& c, uint32_t& d) {
  #pragma HLS INTERFACE ap_ctrl_none port=return
  #pragma HLS INTERFACE s_axilite port=a
  #pragma HLS INTERFACE s_axilite port=b
  #pragma HLS INTERFACE s_axilite port=c
  #pragma HLS INTERFACE s_axilite port=d

  ap_uint<33> x = 0;
  ap_uint<33> y = 0;
  x = a;
  y = b;
  c = x + y;
  d = (x + y) >> 32;
}
```

Po vygenerování IP bloku je zajímavý soubor *xadd_hw.h*. V tomto souboru, který je umístěn na cestě *ip/adder/solution1/impl/misc/drivers/add_v1_0/src* jsou popsány adresy registrů, které lze použít k přístupu ke vstupním a výstupním portům navrženého IP bloku v Python kódu. Soubor se nachází na cestě. Následující komentář je výňatkem z onoho souboru a obsahuje adresy relevantní pro IP blok sčítačky. Všechny registry obsahují 32 bitové hodnoty.Registry a, b jsou vstupní hodnoty, registr c je výsledek součtu a registr d nabývá pouze hodnot 1 nebo 0 podle toho, zda došlo k přetečení součtu přes 32 bitů.

```
// 0x10 : Data signal of a bit 31~0 - a[31:0] (Read/Write)
// 0x18 : Data signal of b bit 31~0 - a[31:0] (Read/Write)
// 0x20 : Data signal of c bit 31~0 - c[31:0] (Read)
// 0x30 : Data signal of d bit 31~0 - c[31:0] (Read)
```

Dalším krokem je vytvoření blokového návrhu v programu Vivado. Pro tento krok je připraven skript spustitelný příkazem `source build_blockdesign.tcl`. Schéma a jeho části lze vidět na obrázku 5.1. Základem schématu je procesní systém ZYNQ7, který je připojen na blok sběrnicového AXI propojení. Na AXI-Lite sběrnici je připojena i sčítačka, se kterou procesní systém může komunikovat. Navíc se ve schématu nachází blok systémového resetu.



Obrázek 5.1: Blokový návrh propojení procesního systému se sčítačkou.

Poslední částí, která předchází použití sčítačky v Python kódu, je vygenerování tří souborů. Pro správnou funkčnost je vyžadováno, aby byly soubory pojmenovány stejným názvem. Příkaz `source build_bistream.tcl` dané 3 soubory vytvoří a zbývajícím krokem je jejich přesunutí do Pynq-Z2. Jedná se o soubory s následujícími příponami:

- adder32bitc.bit
- adder32bitc.tcl
- adder32bitc.hwh

Druhá část se přesouvá z prostředí Vivado nástrojů do Jupyter Notebook a programovacího jazyka Python. Pro práci s overlay je dostupná Python knihovna. Následující 3 příkazy postupně znamenají načtení knihovny, načtení bitstream konfigurace a vypsání informací o dané overlay knihovně, včetně přítomnosti *add* bloku.

```
from pynq import Overlay
overlay = Overlay(<cesta k bit souboru>)
overlay?
```

Přístup k registrům se může provádět dvojím způsobem, pomocí mapovaných jmen registrů nebo s hodnotami adres registrů. O obě tyto metody se stará základní ovladač. Následující první příkaz zobrazí mapovaná jména na registry a pomocí nich pak lze provést výpočet na sčítačce a přečíst výsledek a bit přetečení.

```
overlay.add.register_map
overlay.add.register_map.a = 40
overlay.add.register_map.b = 2
print(overlay.add.register_map.c)
print(overlay.add.register_map.d)
```

Druhou možností s obdobným přístupem je přes adresy registrů.

```
overlay.add.write(0x10, 40)
overlay.add.write(0x18, 2)
print(overlay.add.read(0x20))
print(overlay.add.read(0x30))
```

Ovladač vychází z třídy původního ovladače. V konstruktoru se objevuje položka *description* s mapováním vstupů, přerušením a GPIO piny, které IP blok používá. Parametr *bindto* prováže ovladač s daným blokem v overlay knihovně a následuje samotná funkce *add*, která zapouzdřuje přímé volání registrů.

```
from pynq import DefaultIP
class AddDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:add:1.0']

    def add(self, a, b):
        self.write(0x10, a)
        self.write(0x18, b)
        return self.read(0x20), self.read(0x30)
```

Po aplikaci ovladače a novém načtení bitstream souboru se volání funkce zjednoduší následovně.

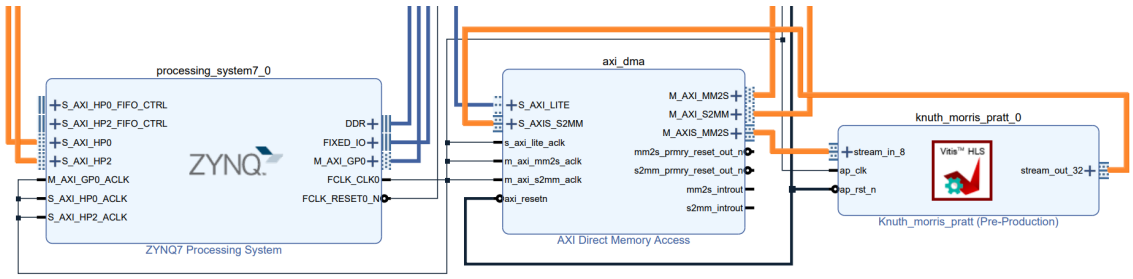
```
x,y = overlay.add.add(40,2)
```

5.2 Implementace akcelerátoru pro vyhledávání v textu

Cílem akcelerátoru pro vyhledávání v textu je spočítat výskyty slov *bear*, *duck* a *shark*, které byly vybrány čistě z ilustrativního hlediska. Kompletní blokový návrh zahrnuje více bloků (například systémový reset a AXI sběrnicové propojení), ale z pohledu akcelerátoru jsou nejdůležitější tři bloky – Procesní systém Zynq, DMA blok a Knuth-Morris-Pratt (KMP) akcelerátor. Tyto bloky lze vidět na obrázku 5.2.

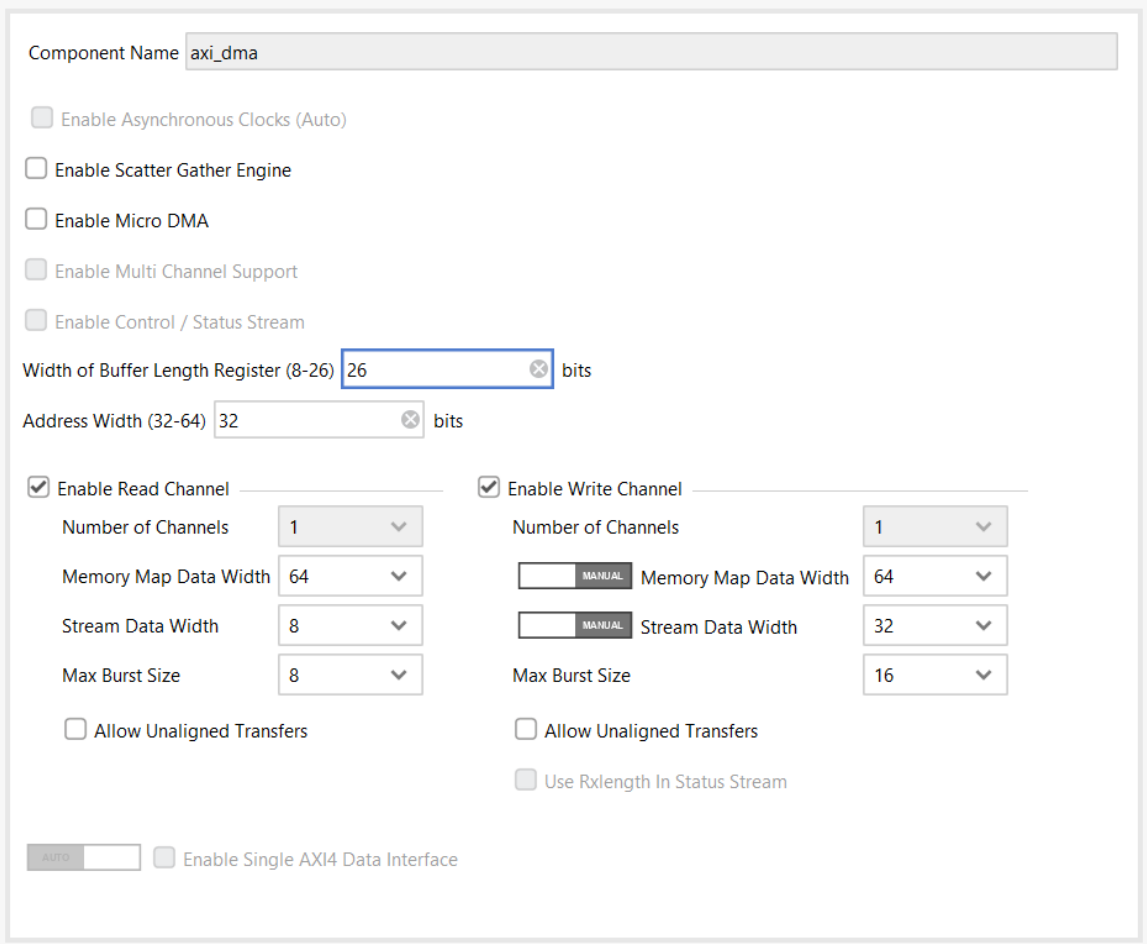
Vlevo se nachází Procesní systém Zynq, který propojuje přes HP (High Performance) porty paměť RAM a DMA blok. HP porty pracují se šířkou dat 64 bitů, která je tak nastavena v základu. Procesní systém má celkem 4 HP porty, přičemž vždy skupina dvou portů (HP0, HP1 a HP2, HP3) sdílí přenosový přepínač pro čtení a zápis. Z pohledu výkonu je tak vhodné (pokud to lze) vybírat porty napříč skupinami. Aktuálně jsou v návrhu použity porty HP0 a HP2.

Propoje vedou skrze sběrnicové AXI propojení, které z důvodu přehlednosti není na obrázku vidět a jsou přivedeny do prostředního DMA bloku na porty M_AXI_MM2S a M_AXI_S2MM. Poslední sekvence názvu portů MM2S (Memory Mapped to Stream) nebo S2MM (Stream to Memory Mapped) určuje směr komunikace z paměti na AXI stream nebo naopak. Port s velmi podobným názvem M_AXIS_MM2S přeposílá načtená data do akcelerátoru, který je na obrázku umístěn vpravo. Zpracovaná data jsou načítána na portu S_AXIS_S2MM.



Obrázek 5.2: Část blokového schéma KMP akcelerátoru.

Pro správnou funkčnost je důležité nakonfigurovat DMA blok [9], jako na obrázku 5.3.



Obrázek 5.3: Část blokového schéma KMP akcelerátoru.

Prvním krokem je zrušení položky „Enable Scatter Gather Engine“. Tato funkce umožňuje sesbírávat a ukládat data jednoho přenosu na různé adresy v paměti RAM, nicméně se jedná o funkci, která v Pynq Z2 není podporována. Další změnou je „Width of Buffer Length Register“. Jedná se o maximální velikost jednoho DMA přenosu. Hodnota 26 opovídá 2^{26} bitů, což je 8 MB. Položky „Memory Map Data Width“ mají v čtecím i zápisovém kanálu velikosti 64 bitů. Jde o základní velikost nastavenou pro Pynq Z2, se kterou pracují HP porty. Tato změna je pevně dána při načítání systému. Změna velikost HP portu na 32 bitů vyžaduje provést restartování Pynq Z2. Předposlední úpravou je nastavení „Stream Data Width“ v čtecím kanálu na 8 bitů (jeden načítaný znak má velikost jednoho bajtu). V zápisovém kanálu má „Stream Data Width“ velikost 32 bitů, protože jsou do paměti zapisovány počty výskytů s datovým typem integer, který má velikost právě 32 bitů. Na závěr je potřeba zrušit nezarovnané přenosy.
Kód samotného akcelerátoru je napsaný v jazyce C a představuje systematicky vytvořený konečný automat. Základem je programová konstrukce *switch*, která na základě hodnoty vybírá odpovídající stav tvořený výrazem *case*. Následující úryvek kódu zobrazuje část pro vyhledání slova *bear*. Lze si vyšimnout, že například načtení chybného znaku *s* ve slově *bear* nevede na návrat do počátečního stavu 0, ale jedná se o první symbol ze slova *shark*, který je tvořen stavem s hodnotou 300. Tento princip je onou klíčovou vlastností algoritmu Knuth-Morris-Pratt. Načítá další symboly, ale nikdy se v textu nevrací zpět.

```
switch (state) {

    case 0: // new symbol
        if (loaded_char.data == 'b') { state = 100; } // b
        else if (loaded_char.data == 'd') { state = 200; } // d
        else if (loaded_char.data == 's') { state = 300; } // s
        else { state = 0; } // other symbols
        break;

    case 100: // b
        if (loaded_char.data == 'e') { state = 101; } // be
        else if (loaded_char.data == 'b') { state = 100; } // b
        else if (loaded_char.data == 'd') { state = 200; } // d
        else if (loaded_char.data == 's') { state = 300; } // s
        else { state = 0; } // other symbols
        break;

    case 101: // be
        ...
        break;

    case 102: // bea
        // bear
        if (loaded_char.data == 'r') { bear_occurrences++; state = 0; }
        ...
        else { state = 0; } // other symbols
        break;
```

Nezbytným krokem použití akcelerátoru v Pythonu je nahrání konfigurace.

```
from pynq import Overlay
image_filter = Overlay(<cesta k bit souboru>)
```

Následuje načtení dat k prohledávání ze souboru *example.txt* v podobě posloupnosti bajtů. Pokud je soubor větší než 8 MB, je potřeba jej rozdělit na menší části, protože maximální velikost jednoho DMA přenosu je právě 8 MB. Příkladový soubor obsahuje 100 odstavců vygenerovaných z webové stránky lorem ipsum [1]. Do tohoto textu bylo na náhodná místa přidáno 21 slov „bear“, 12 slov „duck“ a 7 slov „shark“.

```
file = open("example.txt", "rb")
byte_array = list(file.read())
file.close()
```

Komunikace probíhá skrze DMA IP blok, pro který je nutné předpřipravit podporovanou strukturu dat (numpy array s jednou dimenzí, což je vektor hodnot) a umístit do struktury načtená data ze souboru.

```
from pynq import allocate
import numpy as np

# transform loaded data into numpy array
byte_array = np.array(byte_array)

dma = text_search.axi_dma
dma_send = dma.sendchannel
dma_recv = dma.recvchannel

input_buffer = allocate(shape=(byte_array.shape[0],), dtype=np.uint8)
output_buffer = allocate(shape=(3,), dtype=np.uint32)

# copy loaded data into allocated buffer, that will be send to DMA
input_buffer[:] = byte_array
```

Následuje samotný přenos dat do DMA bloku, který data přepošle do akcelerátoru a sesbírá výsledky s počty výskytů, které jsou následně vypsány.

```
dma_send.transfer(input_buffer)
dma_recv.transfer(output_buffer)

print('"bear" occurrences: ' + str(output_buffer[0]))
print('"duck" occurrences: ' + str(output_buffer[1]))
print('"shark" occurrences: ' + str(output_buffer[2]))
```

Před zahájením dalšího přenosu je vhodné si ověřit, zda předchozí přenos již skončil (následující příkaz má hodnotu True).

```
dma_recv.idle
```

[1] https://lipsum.com/

Zajímavostí při vyhledávání v textu jsou palindromy, tedy slova která se čtou stejně zleva doprava i zprava doleva. Příkladem takového slova je „radar“. Většina vyhledávacích prostředků v počítači vyhledává slova bez překrytí. Pokud je dán řetězec „radaradar“, tak dojde k vyhledání pouze prvního výskytu. Pomocí konečného automatu a Knut-Morris-Pratt přístupu lze vyhledávat i překrývající se sekvence. Výsledkem jsou dva nálezy, **radar**adar a rada**radar**.

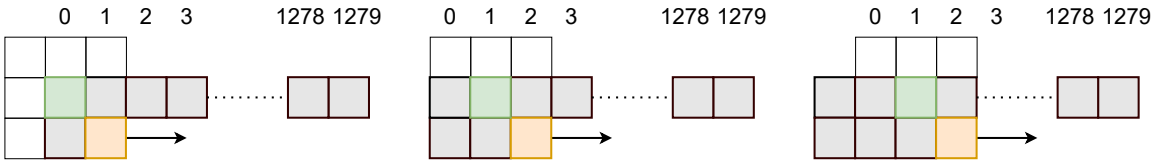
5.3 Implementace obrazového filtru

Implementace je rozdělena na dva samostatné IP bloky, z nichž první řeší barvení pixelů (odstín šedi/červená barva) a druhý se stará o vyhlazování Gaussovým filtrem. Oba akcelerátory jsou navrženy k připojení do zřetězeného zpracování, probíhajícího formou proudové komunikace (stream). V tomto režimu se přenáší data mezi bloky pixel po pixelu. Popis algoritmů v této sekci je postaven na jisté úrovni abstrakce, která by měla vystihnout hlavní myšlenku a princip implementace. Jedná se o výňatek reálné implementace ve zjednodušené formě pseudokódu. U této aplikace také stojí za zmínku soubor s příponou *xdc*, který se nachází ve složce *constraints* (v českém překladu jako omezení). Tento soubor obsahuje definice pro mapování pinů v blokovém návrhu na fyzické piny, které musejí splňovat požadavky na časování a ke kterým jsou fyzicky připojeny HDMI porty na desce.

Obvod pro barvení pixelů postupně načítá 24 bitové pixely. Po načtení pixelu provede výpočet odstínu, jasu a sytosti barvy dle rovnic popsaných v sekci 3.2. Na základě těchto údajů se rozhodne, zda jde o červený pixel nebo jej přepočítá na odstín šedi. Poté je pixel vyslán na výstupní rozhraní proudové komunikace. Tento cyklus se opakuje, dokud není načten „poslední“ pixel. Prakticky je jako poslední pixel označován každý pixel na konci jednoho řádku obrazu, tedy každý 1280 pixel. S příchodem dalších řádků se volání funkce opakuje.

```
red_grey_filter(in_stream, out_stream) {
  while(!last) {
    in_stream.read(pixel);
    last = pixel.last;

    r = pixel.data(7,0);
    g = pixel.data(15,8);
    b = pixel.data(23,16);

    h = compute_hue(r,g,b);
    l = compute_luminosity(r,g,b);
    s = compute_saturation(r,g,b,l);

    if (!red_condition(h,l,s)) {
        pixel = compute_grey(r,g,b);
    }

    out_stream.write(pixel);
  }
}
```

Obvod pro Gaussův filtr rovněž funguje v cyklu, který zpracovává obraz po snímcích. Matice Gaussova filtru vyžaduje znalost 3×3 okolí pixelu, nejdříve se musí přednačíst dostatečný počet pixelů z obrazu. Dostatek znamená načíst celý jeden řádek pixelů o délce *WIDTH* a 1 pixel z druhého řádku. Do této doby nelze zahájit zpracování výstupních pixelů. Od druhého pixelu na 2 řádku lze zahájit zpracovávání. Tato situace spolu s několika následujícími kroky je zobrazena na obrázku 5.4. Oranžovou barvou je zvýrazněn aktuálně načtený pixel a zelenou barvou je označen zpracovávaný pixel.



Obrázek 5.4: Tři kroky posunu okénka Gaussova filtru po obraze.

Projití celého obrazu vyžaduje přečíst $HEIGHT \cdot WIDTH$ pixelů. Po dobu čtení prvního řádku a jednoho pixelu z druhého řádku se neprodukuje žádný výstup a vzniká zpoždění mezi vstupem a výstupem. Celkový počet iterací pro jeden snímek obrazu je dán rovnicí $HEIGHT \cdot WIDTH + (WIDTH + 1)$. V posledních $WIDTH + 1$ iteracích neprobíhá žádné čtení vstupních pixelů a pouze se posílají zpočátku zpožděné pixely.

Paměťový model algoritmu pracuje s uchováváním dvou řádků obrazu, které se cyklicky přepisují s načítáním dalších pixelů. Druhý paměťový prvek tvoří okénko o velikosti 3×3. V každé iteraci se provádí vložení pixelu do paměti řádků. Zároveň se z paměti vytáhne sloupec o velikosti 3 prvků (dva z řádkové paměti a jeden načtený pixel). Následuje posun prvků v okénku o jednu pozici doleva. Jako nový sloupec je vložen ten načtený z paměti.

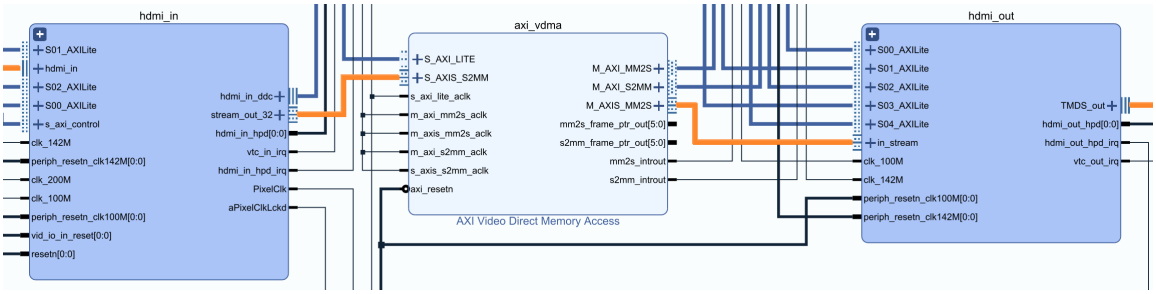
Posledním krokem je kontrola, zda je aktuálně zpracovávaný pixel okrajem obrazu. Když není, provede se operace Gaussova filtru. Obraz má zachovány všechny 3 barevné kanály, tudíž je nutné použít 3 matice paralelně, pro každou barevnou složku (R, G, B) zvlášť.

```
gaussian_filter(in_stream, out_stream) {
  for (i; i < HEIGHT * WIDTH + (WIDTH + 1)) {
    if (i < HEIGHT * WIDTH) {
      in_stream.read(pixel);
    }
    buffer(pixel, column_index, pixels_column[3]);
    shift_window(window[9], pixels_column[3]);

    if (i > IMG_WIDTH) {
        pixel = window[4];
        if (window[4] != border_pixel) {
          pixel = compute_gaussian(window[9]);
        }
        update_counter(filtered_column_index, filtered_row_index);
        out_stream.write(pixel);
    }
    column_index = (column_index == WIDTH - 1) ? 0 : column_index + 1;
  }
}
```

Blokový návrh má základ postavený na base návrhu, jehož soubory jsou dostupné na GitHub stránce [13]. Kompletní blokové schéma je komplexní, ale pro práci s obrazem se lze vymezit pouze na video blok, jehož nejdůležitější části lze vidět na obrázku 5.5. Uvnitř jsou propojeny 3 primární bloky *hdmi_in*, *axi_vdma* a *hdmi_out* a cesta, kterou putují pixely mezi bloky je zvýrazněna oranžově.

Pro zapojení akcelerátoru se nabízejí dvě místa. Buď do výstupní nebo do vstupní části procesu zpracování HDMI signálu. Pokud by byl akcelerátor umístěn na výstup, znamená to, že by obraz musel projít přes výstupní HDMI rozhraní. Musel by tedy být připojen externí monitor. V případě uložení snímku by obraz prošel pouze skrze vstup do řadiče paměti a zcela by se vyhnul projití výstupním blokem se zabudovaným filtrem. Výhodou je, že lze skrze řadič paměti načíst vlastní obraz z paměti a ten poslat přes filtr na výstup.

Druhou a zároveň aktuálně použitou variantou je připojení filtru na blok vstupního zpracování. V tomto případě lze využít HDMI výstup z počítače, provést filtraci ve vstupním bloku a upravený snímek uložit přes řadič paměti do Pynq Z2. Odpadá tak nutnost připojovat externí monitor k prohlédnutí výsledku filtrace. Nevýhodou tohoto zapojení je, že při načtení vstupního obrazu z paměti je snímek poslán na HDMI výstup a vyhne se projitím přes filtr, který je umístěn ve vstupním bloku zpracování signálu.

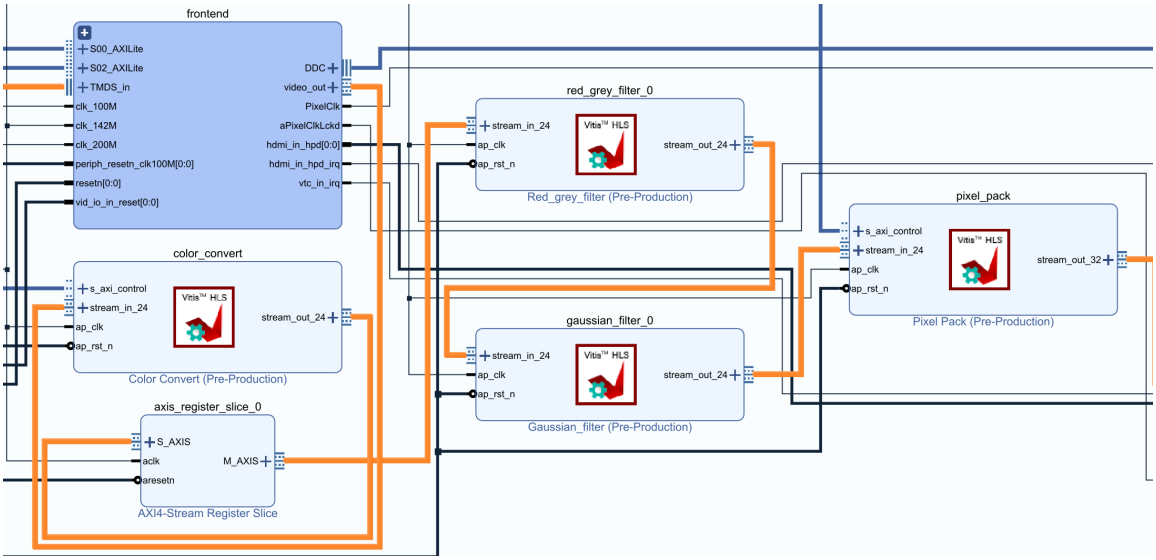


Obrázek 5.5: Nejdůležitější části video bloku.

Rozbor hdmi_in bloku ukazuje obrázek 5.6. Oranžově je vyznačen směr putování pixelů ve zřetězeném zpracování. První blok, nazvaný *frontend*, řeší logiku časování a načítání pixelů z fyzického rozhraní. Načtené pixely přeposílá do bloku *Color Convert*. V základu jsou načítány barevné složky pixelů v pořadí B, G, R. Což pro tento projekt není nijak omezující a tento režim je použit. Avšak v Color Covert bloku jej lze změnit na na pořadí R, G, B. Výstup je vyveden do registru.

Původně následovalo propojení do bloku *Pixel Pack*, ve kterém se vybírá režim zarovnání pixelů na 32 bitovou sběrnici, se kterou dále pracují další bloky a jedná se o standardní zarovnání AXI sběrnice. Zde lze vybrat jeden ze 3 režimů:

- RGBA - umístí jeden pixel do 32 bitů a použije výplň (alfa kanál). Reálná struktura bajtů je [A, B, G, R].
- RGB - umístí 24 bitů z pixelu a do zbylého místa načte data z následujícího pixelu. Opakuje se sled tří 32 bitových přenosů, ve kterém jsou vyznačeny barevné složky a číslo, ke kterému pixelu připadají, [B1, G1, R1, B2], [G2, R2, B3, G3], [R3, B4, G4 R4].
- greyscale - bere pouze jeden kanál v odstínu šedi a do jednoho přenosu umístí 4 pixely [P1, P2, P3, P4].



Obrázek 5.6: Rozbor blokového schématu hdmi_in bloku.

Zapojení akcelerátoru mezi registr a Pixel Pack se jeví jako vhodné, nechává možnost programátorovi určit způsob zarovnávání pro případné další zpracování. Navíc lze v Pythonu změnit i pořadí barevných složek BGR na RGB, čímž se prohodí modrý kanál s červeným a místo ponechání pouze červené barvy dojde k ponechání pouze modré.

Použití akcelerátoru je založeno na knihovních funkcí pro práci s výše zmíněnými bloky v návrhu. Jako první je potřeba nahrát bitstream konfiguraci pro FPGA.

```
from pynq import Overlay
image_filter = Overlay(<cesta k bit souboru>)
```

Druhým krokem je načtení knihovny, provedení počátečního nastavení režimu zarovnání pixelu na 32 bitů a spuštění přenosu HDMI signálu.

```
from pynq.lib.video import *

hdmi_in = image_filter.video.hdmi_in
hdmi_out = image_filter.video.hdmi_out
hdmi_in.configure(PIXEL_RGBA)
hdmi_out.configure(hdmi_in, PIXEL_RGBA)

hdmi_in.start()
hdmi_out.start()
```

Akcelerátor je určen pro rozlišení 1280×720. Aktuálně detekované rozlišení lze zkontrolovat následujícím příkazem, přičemž je možné ověřit i nastavené zarovnání pixelů na 32 bitů.

```
hdmi_in.frontend.mode
hdmi_in.pixel_pack.bits_per_pixel
```

S pomocí knihovny PIL se dá upravený snímek načíst a uložit do paměti v Pynq Z2. Knihovna pracuje s 24 bity na pixel a pokud je použito zarovnávání na RGBA (32 bitů), musí se snímek převést na formát RGB pouze s kanály RGB, což provádí funkce `convert('RGB')`.

Zavoláním proměnné, ve které je snímek uložen, dojde k vykreslení obrázku v Jupyter Notebook.

```
import PIL.Image

frame = hdmi_in.readframe()
img = PIL.Image.fromarray(frame)
img = img.convert('RGB')
img.save("./example.jpg")
img
```

Za zmínku stojí i funkce pro provázání vstupu s výstupem, kdy se načtené snímky z HDMI automaticky přenášejí na výstupní HDMI.

```
hdmi_in.tie(hdmi_out)
```

Pro zajímavost lze zpracovat 60 snímků a přeposílat je na výstup, změřit čas a vypočítat průměrný počet snímků za sekundu. Snímková frekvence by se měla pohybovat v rozemzí 50 - 60 snímků za sekundu (fps), což je plně dostačující například pro úpravu obrazu filmu v reálném čase, jehož snímková frekvence bývá 24 nebo 30 fps.

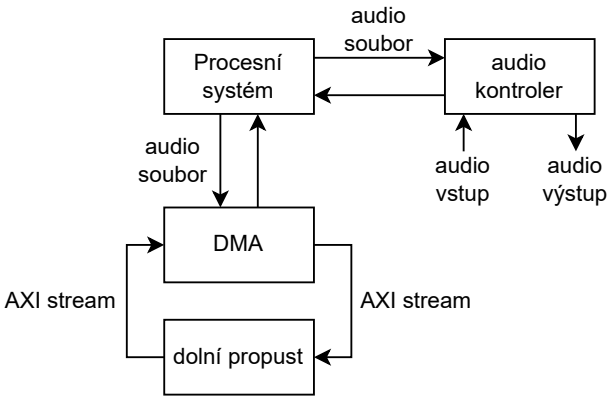
```
import time

numframes = 60
start = time.time()

for _ in range(numframes):
    f = hdmi_in.readframe()
    hdmi_out.writeframe(f)

end = time.time()
print("Frames per second: " + str(numframes / (end - start)))
```

5.4 Implementace dolnopropustního audio filtru

Návrh aplikace dolnopropustního filtru ze sekce 4.4 musel být upraven, neboť zvukový řadič neumožňuje přístup k jednotlivým vzorkům v reálném čase. Nový návrh z obrázku 5.7 nejdříve provádí převod audio souboru přes filtr dolní propusti a následně lze nový soubor poslat do audio řadiče, který řeší přístup k audio vzorkům ve vlastní režii.

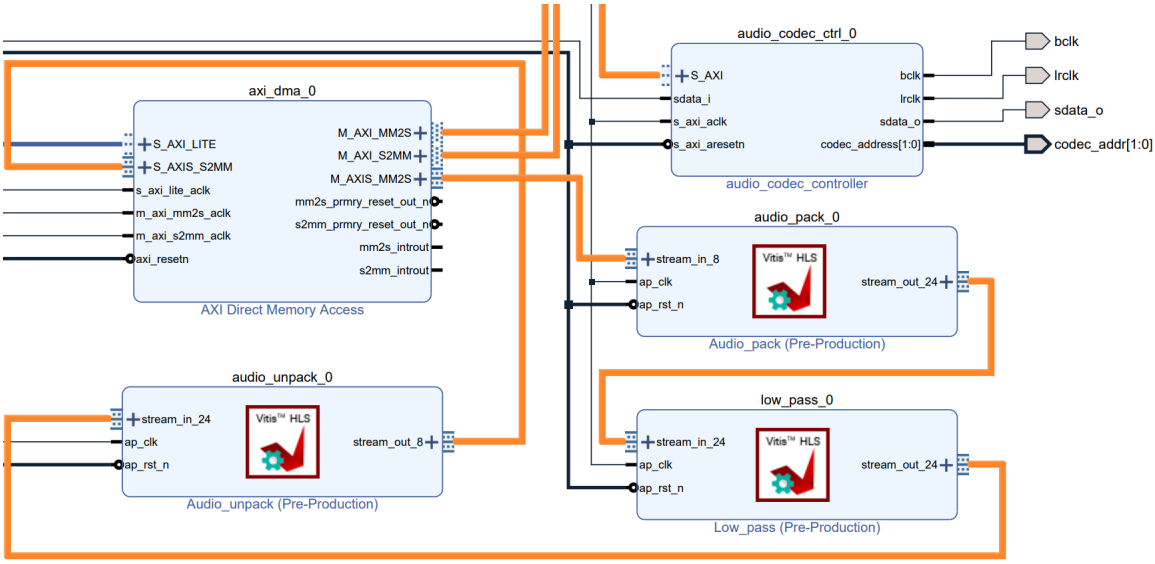
Řadič neposkytuje ani žádné pokročilé funkce, takže nelze přehrávání přerušit, ani upravit hlasitost. Soubor se musí prostřednictvím kontroleru nejdříve nahrát do kodeku a poté spustit přehrávání. Narhávání zvuku ze vstupu musí být předem vyhrazeno dobou (počtem sekund). Kodek opět provádí ukládání dat ve vlastní režii s pevně danou vzorkovací frekvencí, nastavenou na 48 kHz.

V blokovém návrhu je zvukový řadič propojen přes AXI sběrnici s procesním systémem Zynq. Dále je k procesnímu systému připojen DMA blok, který přenáší data mezi pamětí RAM a akcelerátorem s funkcí dolní propusti. Akcelerátor není napojen na DMA blok napřímo, protože podporované velikosti DMA přenosů jsou 8, 16, 32 bitů, zatímco pro akcelerátor je potřeba přenášet 24 bitové vzorky.



Obrázek 5.7: Nový návrh blokového schéma audio filtru.

DMA blok je tak nastaven na přenos 8 bitů a jsou použity pomocné bloky. Vstupní blok načítá 3 bajty a seskupuje je do jedné 24 bitové hodnoty a naopak výstupní blok rozkládá 24 bitové vzorky na tři bajty. Vyfiltrované hodnoty se ukládají do paměti a výsledný soubor lze poslat do zvukového řadiče. Následující obrázek 5.8 obsahuje pouze důležité bloky se zvýrazněným propojením mezi nimi, nejedná se o kompletní blokové schéma. Podrobný návod ke konfiguraci DMA bloku lze nalézt v příkladu s vyhledáváním textu 5.2.



Obrázek 5.8: Část schématu akcelerátoru dolní propusti.

Implementace bloku akcelerátoru v jazyce C pracuje se 3 poli. V prvním jsou uloženy koeficienty filtru a další dvě pole slouží pro ukládání dat z levého a pravého kanálu stereo audia. Ve WAV souboru jsou data z obou kanálů proložena, tj. jeden načtený vzorek je z levého kanálu, druhý z pravého. . . Po načtení každého vzorku se musí překlopit zpracovávaný kanál. Načtené vzorky v poli z předchozího kroku jsou posunuty, je do nich vložen nový vzorek, následuje provedení konvoluce přes celé pole. Na závěr je výsledná hodnota odeslána zpět do DMA jednotky.

Na vstupu přichází 24 bitové celočíselné hodnoty, které se musejí převést na desetinná čísla z rozsahu -1.0 až 1.0, protože koeficienty jsou v desetinné formě. Za tímto účelem se

vstupní vzorek dělí hodnotou 2^{23}, polovinou rozsahu 2^{24} (polovina záporných a polovina kladných hodnot). Na výstupu je vzorek násoben 2^{15}, polovinou rozsahu 16 bitových hodnot. Je tak reflektován přístup ze software Octave[2], který ukládá 16 bitové vzorky. Audio řadič ale vyžaduje na vstupu 24 bitové vzorky, a hodnota je tak dále vynásobena 10, tím dojde ke zvýšení amplitudy 24 bitového vzorku, aby přibližně odpovídala hlasitosti 16 bitového vzorku.

```
void low_pass(in_stream in_sample, out_stream out_sample) {
    static float coef_arr[69] = {0.0016520, 0.0017268, ...};
    static float left_ch[69] = {0, 0, 0, ...};
    static float right_ch[69] = {0, 0, 0, ...};
    static bool is_left_channel = true;

    read_wav_sample(in_sample);
    in_sample = in_sample / 8388608;

    if (is_left_channel == true) {
        shift(left_ch);
        insert(left_ch, in_sample);

        for (int i = 0; i < 69; i++) {
            out_sample += (left_ch[i] * coef_arr[i]);
        }
    }
    else {
        shift(right_ch);
        insert(right_ch, in_sample);

        for (int i = 0; i < 69; i++) {
            out_sample += (right_ch[i] * coef_arr[i]);
        }
    }
    out_sample = (out_sample * (32768 * 10))
    is_left_channel = !is_left_channel;
    write_wav_sample(out_sample);
}
```

Prvním krokem použití akcelerátoru v Pythonu je nahrání konfigurace a vytvoření odkazů k IP blokům. Následující úryvky kódů nejsou kompletní a jde pouze o ilustraci hlavní myšlenky.

```
audio_filter = Overlay("./audio_filter.bit")
dma = audio_filter.axi_dma_0
dma_send = dma.sendchannel
dma_recv = dma.recvchannel
audio_ctrl = audio_filter.audio_codec_ctrl_0
```

Pokud existuje soubor s již dříve upraveným audio signálem, je potřeba ho smazat.

[2] https://www.gnu.org/software/octave/

```
if (os.path.isfile("./filtered_example.wav")):
    os.remove("./filtered_example.wav")
```

Načtení a zkopírování WAV hlavičky, která má velikost 44 bajtů do nového souboru, kde bude umístěn filtrovaný audio signál. Z hlavičky souboru je extrahována informace o velikosti zvukových dat, které následují hned za hlavičkou.

```
original_audio = open("./example.wav", "rb")
filtered_audio = open("./filtered_example.wav", "ab")
wav_header = original_audio.read(44)
filtered_audio.write(wav_header)
data_length = int.from_bytes(wav_header[40:44], "little")
```

DMA blok podporuje maximální velikost 8 MB jednoho přenosu. Soubory WAV nejsou nijak komprimovány a v praxi je velikost audio souboru mnohdy vyšší. Z tohoto důvodu je vytvořena smyčka, která čte a zpracovává soubor postupně po 8 MB blocích. Poslední načtený blok nemusí být kompletní a již může obsahovat část s metadaty, kterou je nutné odseknout a neposílat do akcelerátoru.

```
actual_length = 0
meta_data = None
for i in range(0, data_length, 8388608):
    audio_data = read_original_audio(8388608)
    actual_length = actual_length + len(audio_data)

    # cut metadata from last 8 MB chunk
    if (actual_length > data_length):
        cut_data(meta_data, audio_data)

    input_buffer = allocate(shape=(audio_data.shape[0],), dtype=np.uint8)
    output_buffer = allocate(shape=(audio_data.shape[0],), dtype=np.uint8)
    input_buffer[:] = audio_data
    do_dma_transfer(input_buffer, output_buffer)
    filtered_audio.write(output_buffer)

    # save metadata to file
    if (actual_length > data_length):
        filtered_audio.write(meta_data)

    free(input_buffer, output_buffer)
```

Soubor může obsahovat další objemné metadata (přidané texty písně s časovými stopami, obrázky alba ve vysoké kvalitě, obrázky umělce...) a v zájmu zachování i těchto dat se musí překopírovat zbytek původního souboru do nového.

```
rest_of_metadata = original_audio.read()
filtered_audio.write(rest_of_metadata)
original_audio.close()
filtered_audio.close()
```

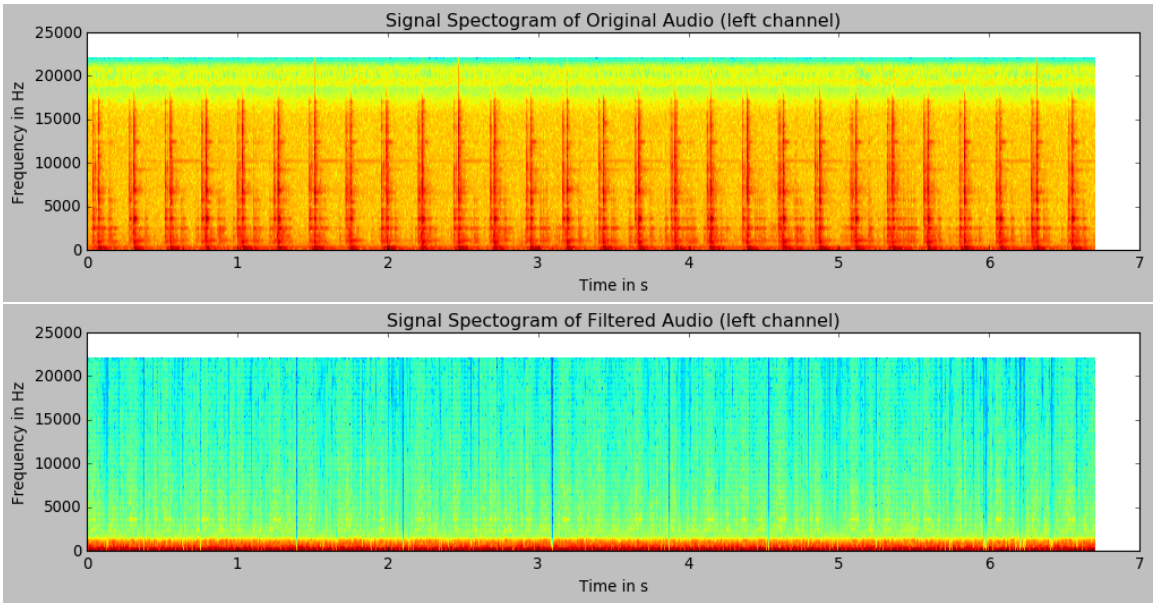
Zvukovou stopu jde přehrát přímo v prostředí Jupyter Notebook.

```
from IPython.display import Audio as IPAudio
IPAudio("./filtered_example.wav")
```

A lze ji poslat i do zvukového řadiče v FPGA.

```
audio_ctrl.load("./example.wav")
audio_ctrl.play()
```

Závěrem je poskytnut pohled na dvě frekvenční charakteristiky zvukové stopy [10], která byla převedena do formátu použitelného pro audio kontrolér (převzorkování z 48000 na 44100 vzorků za sekundu a úprava z mono na stereo kanály). Horní část obrázku 5.9 je spektogram původní zvukové nahrávky s frekvencemi cca do 18 kHz a ve spodní části se nachází spektogram vyfiltrované nahrávky s frekvencí do cca 1 kHz.



Obrázek 5.9: Spektogramy původní a vyfiltrované nahrávky.

5.5 Shrnutí výsledků

V rámci této kapitoly byly implementovány aplikace sčítačka, vyhledávání v textových souborech, filtrace obrazových a zvukových dat. Každá implementace je provedena v rámci Jupyter Notebook dokumentu spolu s teorií potřebnou k zasazení do problematiky. Aplikace byly testovány přímo na platformě Pynq Z2.

Filtrace obrazu zvládá zpracovávat přibližně 55.421 snímků za sekundu s rozlišením 1280×720. Ukládání každého snímku na SD kartu je omezeno rychlostí zápisu. Při pokusu uložit 60 snímků proces trval 8.235, to ve výsledku dělá pouze 7.286 zpracovaných snímků za sekundu.

Vyhledávání v textu bylo otestováno na 100 odstavcích vygenerovaného lorem ipsum[3] textu. Do tohoto textu bylo náhodně umístěno 21 slov „bear“, 12 slov „duck“ a 7 slov „shark“. Celková velikost souboru činí 62688 bajtů (62 kB). Implementace na FPGA vrátila

[3] https://lipsum.com/

výsledky za 5.35 ms, zatímco čistě softwarovému řešení trval úkol 6.75 ms. FPGA implementace je ve výsledku rychlejší přibližně o jednu čtvrtinu rychlejší.

Zvukový filtr je poměrně komplikované otestovat na rychlost, protože softwarová část řeší načítání souboru a vyseknutí dat vzorků zvuku a kopii metadat. FPGA část řeší zpracování vzorků. Nejedná se tedy o čistě HW řešení. Nicméně otestován byl celý proces (SW i HW část) na vzorku audia trvajícího 6.7 s. Parametry nahrávky jsou 2 kanály, 24 bitů na vzorek a vzorkovací frekvence 44100 Hz. Kompletní filtrace trvala 2.61613 s. Z čehož vyplývá, že kdyby byl dostupný pokročilý zvukový ovladač umožňující signál zpracovávat v reálném čase, tak platforma Pynq Z2 by v tomto úkolu obstála dobře.

Kapitola 6

Závěr

Cílem diplomové práce bylo navrhnout a implementovat 3 až 5 aplikací využívajících FPGA jednotku a klíčové komponenty platformy Pynq Z2. Implementaci řádně zdokumentovat a vytvořit podpůrné materiály v podobě dokumentů Jupyter Notebook, které zájemce o platformu Pynq Z2 uvedou do problematiky a pomohou jim jednoduchou modifikací vytvořených aplikací dosáhnout daného cíle. Práce zahrnuje návrh pěti aplikací z oblastí vyhledávání v textu, filtrování obrazu, zpracování audio signálu a klasifikace internetových paketů, přičemž byly implementovány 4 aplikace.

První aplikací je nenáročná sčítačka. Předností této aplikace je seznámení s nástroji a vyřešení prvotních problémů k úspěšnému provedení syntézy, sestavení aplikace a přístupu z procesního systému do IP bloku v FPGA skrze GP (General Purpose) porty.

Druhou aplikací je obrazový filtr v reálném čase. Přiblížen je proces zřetězeného zpracování pixelů, význam a činnost hlavních bloků, které se na zpracování podílejí a jejich konfigurace. Do zřetězeného zpracování jsou zakomponovány dva akcelerátory. Jeden pracuje se samotnými pixely a řeší jejich barvení (buď odstín šedi nebo červená barva), druhý akcelerátor je implementací konvoluční 2D masky v podobě Gaussova filtru. Základní Gaussův filtr pracuje s okolím o velikosti 3×3 pixelů a v akcelerátoru se nachází efektivní mechanismus ukládání okolí pouze pomocí 2 řádkového pole. Okraje obrazu nejsou filtrovány, neboť jim chybí někteří sousedé, ale představeny jsou i další mechanismy jak řešit krajní body. Celý řetězec zpracování pracuje s načítáním pixelů sekvenčně jeden za druhým (stream). Při testování bylo dosaženo výkonu 50 až 57 snímků za sekundu v HD rozlišení 1280×720.

Třetí implementovanou aplikací je vyhledávání v textu se zaměřením na správnou konfiguraci DMA přenosu dat mezi akcelerátorem a RAM pamětí. Úskalím DMA bloku je maximální velikost jednoho přenosu omezená na 8 MB dat. Zvolený algoritmus Knuth-Morris-Pratt je založen na konečném automatu a umožňuje snadno vyhledávat i sled překrývajících se palindromů (slov, která se čtou zleva doprava a zprava doleva stejně). Při vyhledávání v textu je výzvou i kódování znaků. Data jsou načítána bajt po bajtu a při načítání složitějších znaků, například s diakritikou, je potřeba jeden stav konečného automatu rozdělit na více stavů, kde každý odpovídá jednomu bajtu.

Poslední realizovanou aplikací je filtrace digitálního audio signálu přes dolní propust. Aplikace předvádí přístup a práci se vzroky zvukového formátu WAV a poukazuje na výrazné nedostatky dostupného zvukového řadiče, který znemožňuje práci se signálem v reálném čase. Není možné ani dynamicky měnit hlasitost nebo zastavit přehrávaná data. V průběhu zpracování zvuku dochází ke mnoha nepřesnostem jako je zaokrouhlování hodnot, velikost datových tipů (float a double), nevyváženost počtu kladných a záporných čísel,

kvalita nahrávek, možnosti reproduktorů... Tyto a mnoho dalších parametrů ovlivňuje výsledný signál.

V rámci návrhu páté aplikace jsem nastudoval základní princip klasifikace paketů s použitím prosévání paketů přes rozhodovací strom spolu s položkami, které lze v internetovém paketu využít ke klasifikaci.

Implementované aplikace byly vybírány s ohledem zastřešit použití hlavních klíčových prvků. Aplikace mají nejen edukativní význam, ale jejich snadnou úpravou lze dosáhnout například omezením modré barvy v obraze, transformovat obraz pro hranovou detekci či barevně odlišených vzorů, skrytí nevhodných slov textu za znaky „*“.

Navázát lze na práci ve směru implementace dvou zbylých navržených aplikací, využít již představené principy a rozšířit je o dynamickou konfiguraci například hodnot masky 2D konvoluce. Nabízí se i vytvoření zcela nových a komplexních aplikací. Prozkoumat další pokročilé a sofistikované metody vyhledávání v textu, u kterých bude možné dynamicky měnit počet a délky vyhledávaných slov. Další oblastí může být zpracování časových řad v reálném čase, kde je možné zjišťovat statistické vlastnosti, odhalovat hodnoty způsobující anomálie, doplňovat chybějící hodnoty, provádět agregaci hodnot nebo predikovat budoucí hodnoty za účelem analýzy chování systému. S tímto tématem souvisí i akcelerace neuronových sítí na FPGA. Spolu s dalšími aplikacemi tak lze vytvořit robustní základnu aplikací a manuálů, která by byla postupně rozšiřována o nové znovuvyužitelné a dobře zdokumentované aplikace.

Literatura

[1] Arduino. *Arduino* [online]. 2022 [cit. 2022-1-24]. Dostupné z: https://www.arduino.cc/.

[2] Bennett, C. L. *Digital Audio Theory: A Practical Guide*. 1. vyd. Routledge, 2021. ISBN 978-0-367-27655-3.

[3] Deeirng, S. *Internet Protocol, Version 6 (IPv6)* [online]. 1998 [cit. 2022-1-16]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc2460.

[4] Espressif. *Espressif Systems (Shangai) CO., LTD.* [online]. 2021 [cit. 2021-12-26]. Dostupné z: https://www.espressif.com/en.

[5] Flynn, M. J. a Luk, W. *Computer System Design: System-on-Chip*. 1. vyd. John Wiley & Sons, Inc., 2011. ISBN 9781118009901.

[6] Information Sciences Institute, University of Southern California. *Internet Protocol* [online]. 1981 [cit. 2021-12-31]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc791.

[7] Intel Corporation. *Chipset and System-on-a-Chip (SoC) Refrence for Intel NUC* [online]. Intel, listopad 2021 [cit. 2021-12-26]. Dostupné z: https://www.intel.com/content/www/us/en/support/articles/000056236/intel-nuc.html.

[8] Limited, A. *AMBA AXI and ACE Protocol Specification* [online]. 2021 [cit. 2022-01-24]. Dostupné z: https://developer.arm.com/documentation/ihi0022/latest/.

[9] McCabe, C. *PYNQ DMA tutorial (Part 1: Hardware design)* [online]. Prosinec 2021 [cit. 2022-05-11]. Dostupné z: https://discuss.pynq.io/t/tutorial-pynq-dma-part-1-hardware-design/3133.

[10] Naro. *Timer 2* [online]. 2005 [cit. 2022-05-16]. Dostupné z: https://bigsoundbank.com/detail-0082-timer-2.html.

[11] Papp, D. B. *Let's do Color & Math* [online]. Medium, 2017 [cit. 2022-01-26]. Dostupné z: https://donatbalipapp.medium.com/colours-maths-90346fb5abda.

[12] Pynq. *Pynq - Python productivity for Zynq* [online]. 2021 [cit. 2021-12-26]. Dostupné z: http://www.pynq.io/.

[13] PYNQ. *Xilinx/PYNQ: Python productivity for ZYNQ* [online]. PYNQ, 2022 [cit. 2022-01-26]. Dostupné z: https://github.com/Xilinx/PYNQ.

[14] Raspberry Pi. *Raspberry Pi* [online]. 2021 [cit. 2021-12-25]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[15] Sapp, C. *WAVE PCM soundfile format* [online]. 1997 [cit. 2022-05-16]. Dostupné z: http://soundfile.sapp.org/doc/WaveFormat/.

[16] Saravanan, C. *Color Image to Grayscale Image Conversion* [online]. 2010 [cit. 2021-12-31]. Dostupné z: https://www.researchgate.net/publication/224130500_Color_Image_to_Grayscale_Image_Conversion.

[17] Stephen, G. A. *String Searching Algorithms*. 1. vyd. World Scientific Publishing Co. Pte. Ltd., 1994. ISBN 981-02-1829-X.

[18] study-ccna.com. *Ethernet frame* [online]. study-ccna.com, 2022 [cit. 2022-01-26]. Dostupné z: https://study-ccna.com/ethernet-frame/.

[19] Technology Un-Limited. *Introducing TUL PYNQ™-Z2* [online]. 2020 [cit. 2021-12-27]. Dostupné z: https://www.tulembedded.com/FPGA/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf.

[20] Xilinx. *Python productivity for Zynq* [online]. PYNQ, 2018 [cit. 2022-01-26]. Dostupné z: https://pynq.readthedocs.io/en/v2.4/pynq_libraries/video.html.

[21] Xilinx. *Zynq-7000 SoC Data Sheet: Overview* [online]. 2018 [cit. 2021-12-26]. Dostupné z: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[22] Xilinx. *Video* [online]. 2021 [cit. 2022-1-16]. Dostupné z: https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries/video.html.

[23] Xilinx. *Zynq-7000 SoC Technical Reference Manual* [online]. 2021 [cit. 2021-12-26]. Dostupné z: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.