VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

UKÁZKY HARDWAROVÉ AKCELERACE NA PŘÍPRAVKU PYNQ Z2
HARDWARE ACCELERATION DEMO ON THE BOARD PYNQ Z2

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PAVEL VOSYKA**
AUTHOR

VEDOUCÍ PRÁCE — **Doc. Ing. JAN KOŘENEK, 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: **Vosyka Pavel, Bc.**
Program: Informační technologie
Obor: Vestavěné systémy
Název: **Úkázky hardwarové akcelerace na přípravku Pynq Z2**
Hardware Acceleration Demo on the Pynq Z2 Board
Kategorie: Počítačová architektura
Zadání:

1. Seznamte se s přípravkem Pynq Z2 a vývojovými nástroji pro technologii Xilinx Zynq.
2. Na základě konzultace s vedoucím navrhněte sadu tří úloh zaměřených na oblast hledání řetězů, práci s maticemi a na klasifikaci dat pomocí rozhodovacích stromů. Úlohy musí být navrženy tak, aby ukazovaly přínos hardwarové akcelerace na přípravku Pynq Z2.
3. Vytvořte vzorové implementace vybraných úloh tak, aby bylo možné prostřednictvím parametrů nastavit míru paralelního zpracování (zrychlení) a množství zabraných hardwarových zdrojů.
4. Vzorové implementace úloh ověřte v simulacích a na přípravku Pynq Z2. Současně zhodnoťte dosaženou míru zrychlení a množství zabraných zdrojů.
5. K jednotlivým úlohám vytvořte krátký návod tak, aby si studenti mohli vytvořené implementace vyzkoušet v simulacích, a na přípravku Pynq Z2 a případně si mohli zdrojové kódy i mírně modifikovat.
6. V závěru diskutujte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2 zadání.

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

Vedoucí práce: **Kořenek Jan, doc. 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 hardwarovou akcelerací na platformě Pynq Z2 osazenou technologií Xilinx Zynq. Na této platformě byly navrženy tři úlohy demonstrující hardwarovou akceleraci. Primárním cílem úloh bylo prezentovat hardwarovou akceleraci pro výukové účely, proto byla snaha je vytvořit co nejjednodušeji, aby byly dobře pochopitelné. Hardwarové akcelerátory jsou napsány v jazyku VHDL a jejich obsluha je zajištěna pomocí aplikace v Pythonu v rámci technologie Pynq. Všechny úlohy byly implementovány a ověřeny na dostupném hardwarovém přípravku.

Abstract

The work deals with a hardware acceleration on the Zynq platform with Pynq technology. Three examples demonstrating hardware acceleration were designed for teaching purposes. The effort was to make examples as simple as possible to make them easy to understand. Hardware accelerators are implemented in VHDL language and driven by implemented Python application. The examples were successfully implemented and evaluated.

Klíčová slova

Hardwarová akcelerace, Pynq, Pynq Z2, Zynq, Xilinx

Keywords

Hardware acceleration, Pynq, Pynq Z2, Zynq, Xilinx

Citace

VOSYKA, Pavel. *Ukázky hardwarové akcelerace na přípravku Pynq Z2*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jan Kořenek, Ph.D.

Ukázky hardwarové akcelerace na přípravku Pynq Z2

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Pavel Vosyka
16. května 2022

Poděkování

Děkuji Doc. Ing. Janu Kořenkovi, Ph.D. za vedení práce.

Obsah

Kapitola 1

Úvod

Informační technologie jsou dnes nedílnou součástí naší společnosti. Od chytrých telefonů a sociálních sítí až po předpověď počasí a industriální aplikace. Informační technologie lze nalézt u většiny lidských aktivit. V současné době se jedná o široký a rychle rostoucí obor, který se velmi pravděpodobně bude v dohledné budoucnosti dále zvětšovat a je možné, že tyto technologie budou postupně potřebné v každém oboru lidské činnosti.

S tím je spojený velký nárůst dat, který se zvyšuje i díky rozšiřující se digitalizaci. Řeší se řady výpočetně náročných úloh z různých oblastí informačních technologií a tyto data je potřeba zpracovávat. To se děje v datových centrech. Příkladem mohou být úlohy z oblasti síťové bezpečnosti, počítačového vidění a umělé inteligence. Datová centra mají velkou spotřebu a výkonu se dosahuje velkým počtem serverů. Softwarové řešení je u náročných úloh často příliš pomalé, a proto je třeba výpočet urychlit pomocí hardwarové akcelerace, která nejen urychluje výpočty, ale může i snížit spotřebu. Řada akceleračních technologií se začíná stále častěji používat, a právě proto v praxi vzniká nedostatek kvalifikovaných lidí, kteří se hardwarovou akcelerací zabývají. Mezi populární technologie pro akceleraci patří FPGA, akcelerace na grafické kartě, ale i další technologie. Akcelerace se používá také ve vestavěných systémech, jako je vyhodnocování dat pro autonomní řízení aut nebo dronů, nebo třeba ke zpracování dat z kamerových systémů.

Cílem mojí práce je vytvořit tři úlohy, které budou sloužit jako ukázky hardwarové akcelerace studentům. Tyto úlohy demonstrují možnosti zrychlení výpočtu pomocí hardwaru a slouží také jako ukázka programování platformy Pynq. První úloha se zabývá vyhledáváním v textu. Jejími vstupy jsou vyhledávané slovo a samotný text, ve kterém se slovo hledá. Výsledkem vyhledávání je pozice nalezeného slova v textu, nebo informace, že se slovo v textu nenalézá. Druhá úloha řeší klasifikaci dat, jedná se o rozdělení dat do tříd podle zadaných pravidel. Vstupem úlohy jsou třídicí pravidla a samotná data. Výsledkem klasifikace je třída, do které vstupní data patří. Poslední úlohou je obrazový filtr. Jde o mediánový filtr, který se používá k odstranění šumu ve fotkách a videích.

V kapitole 2 je popsána technologie Zynq, včetně procesoru a programovatelné logiky. Dále jsou popsány vývojové nástroje, které lze použít pro vývoj akcelerátoru na platformě Zynq. V další části je představena technologie Pynq a oblasti hardwarové akcelerace. V kapitole 3 je shrnutí požadavků na návrh úloh. Jsou zde popsány obecné požadavky na všechny úlohy a důvody pro některá rozhodnutí při návrhu. V kapitole 4 jsou úlohy detailně specifikovány a navrženy. A v kapitole 5 jsou představeny a zhodnoceny výsledky práce.

Kapitola 2

Teoretický rozbor

Vestavěná zařízení se často vytváří na bázi vestavěných procesorů, nejčastěji z rodiny ARM. Procesory slouží k ovládání a řízení systémů, ale i k realizaci klíčových algoritmů. Díky nízkému příkonu procesoru je ale těžké dosáhnout rychlého zpracování dat. Pro zrychlení zpracování dat se proto často část algoritmů akceleruje v hardware. Existují také dedikované vestavěné procesory, které jsou uzpůsobené k řešení určité úlohy nebo třídy úloh. Kromě optimalizované architektury mají fixní hardwarové akcelerátory pro řešení těchto úloh. Dále existuje technologie FPGA, která umožňuje implementovat prakticky libovolný hardwarový akcelerátor. Tuto technologii používá právě technologie Zynq. Zynq je SoC (System on Chip) kombinující programovatelnou logiku (FPGA) s procesorem. Tato kapitola popisuje technologii Zynq. Je zde představena procesorová jednotka a především také programovatelná logika. Ta je realizována technologií FPGA. Dále je zde popsána technologie Pynq, která zjednodušuje práci s platformou Zynq. Pynq podporuje řadu vývojových desek, tato práce je zaměřena na přípravek Pynq Z2.

2.1 Technologie Xilinx Zynq

Technologie Xilinx Zynq v sobě integruje FPGA a ARM procesor v rámci jednoho systému na čipu (SoC). Xilinx momentálně nabízí dvě rodiny produktů této techonologie, Zynq-7000 a Zynq-7000S. Zynq integruje výpočetní systém (Processing System, dále jen **PS**) a programovatelné hradlové pole (Programmable Logic, dále jen **PL**). PS obsahuje dvoujádrový procesor ARM Cortex-A9 s frekvencí až 1 GHz, v případě Zynq-7000S pouze jednojádrový[4]. Součástí PS jsou kromě ARM procesoru také paměti a propojovací logika. PL integruje programovatelnou logiku založenou na technologii FPGA Artix-7 nebo Kintex-7. Parametry programovatelné logiky se zásadně liší podle verze.

Klíčovou součástí technologie Xilinx Zynq je procesor, který je tvořen jedním nebo dvěma jádry ARM Cortex-A9 pracující na frekvenci od 766MHz do 1 GHz. Maximální frekvence procesoru záleží na konkrétním produktu. Procesor je superskalární, má plnou podporu virtuální paměti a dynamickou predikci skoků. Instrukce se zpracovávají mimo pořadí (out of order) a jádro dokáže dekódovat dvě instrukce každý takt. Procesor má rozšíření o SIMD (single instruction multiple data) instrukce, rozšíření je označované jako Neon nebo také Advanced SIMD. Toto rozšíření je povinné u procesorů Cortex-A8, ale u Cortex-A9 procesorů je volitelné. Neon zajišťuje efektivní akceleraci při zpracování multimediálních dat a jiných signálů. Kromě toho procesor může být přepnut do stavu, ve kterém vykonává Java Byte Code.

Každé jádro má vždy dvě L1 cache, jednu instrukční a jednu datovou. Obě cache mají velikost 32 KB. ARM instrukce jsou 32 bitové a musí být v paměti zarovnány na slova. V případě jiných instrukcí, které procesor také podporuje, jako jsou Thumb instrukce o délkách 16 a 32 bitů, je vyžadováno zarovnání na půlslova. Data takto zarovnána být nemusí, ale při nezarovnaném přístupu do paměti je přístup rozdělen na dva zarovnané přístupy a data jsou automaticky spojena. Operace je tak pomalejší.

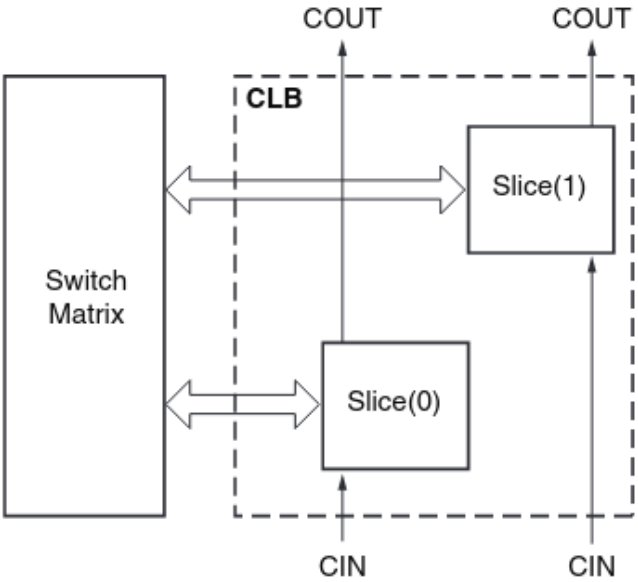
L2 cache má velikost 512 KB a je sdílená mezi jádry. Do L2 má také přístup programovatelná logika (PL). L3 cache už přítomná není. Vedle L2 je také OCM (on chip memory), která poskytuje krátkou odezvu a velikost 256 KB. K systému lze připojit i externí DDR paměť. Podporované standardy jsou DDR3, DDR3L, DDR2 a LPDDR2.

Programovatelná logika dokáže díky paralelnímu zpracování urychlit definovaný algoritmus nebo jeho část oproti procesoru. V PL lze totiž realizovat výpočet paralelně. Vzhledem k možnosti paralelizace dané úlohy a počtu dostupných zdrojů PL lze dosáhnout velkého zrychlení oproti výpočtu na procesoru. PL je výhodná i pro některé časově kritické aplikace. Urychlení výpočtu není jedinou výhodou PL. Programovatelná logika dokáže vykonávat definovaný algoritmus nebo jeho část místo procesoru. To odstraní zátěž z procesoru a umožní mu se věnovat jiným věcem. Vykonávaný algoritmus se může za běhu i měnit, jelikož je PL plně programovatelná.

PL technologie Xilinx Zynq je založena na technologii FPGA Artix-7 nebo Kintex-7. Kintex je výkonnější technologie a je přítomna na největších verzích platformy Zynq, konkrétně 7z030, 7z035, 7z045 a 7z100. Technologie Artix je menší a je zaměřena spíše na levnější a menší obvody. Programovatelná logika zabírá podstatnou část čipu. Jelikož se procesor nalézá na stejném čipu jako PL, má k logice dobrý přístup. Procesor má s logikou rychlou komunikaci a může logiku kdykoli rekonfigurovat.

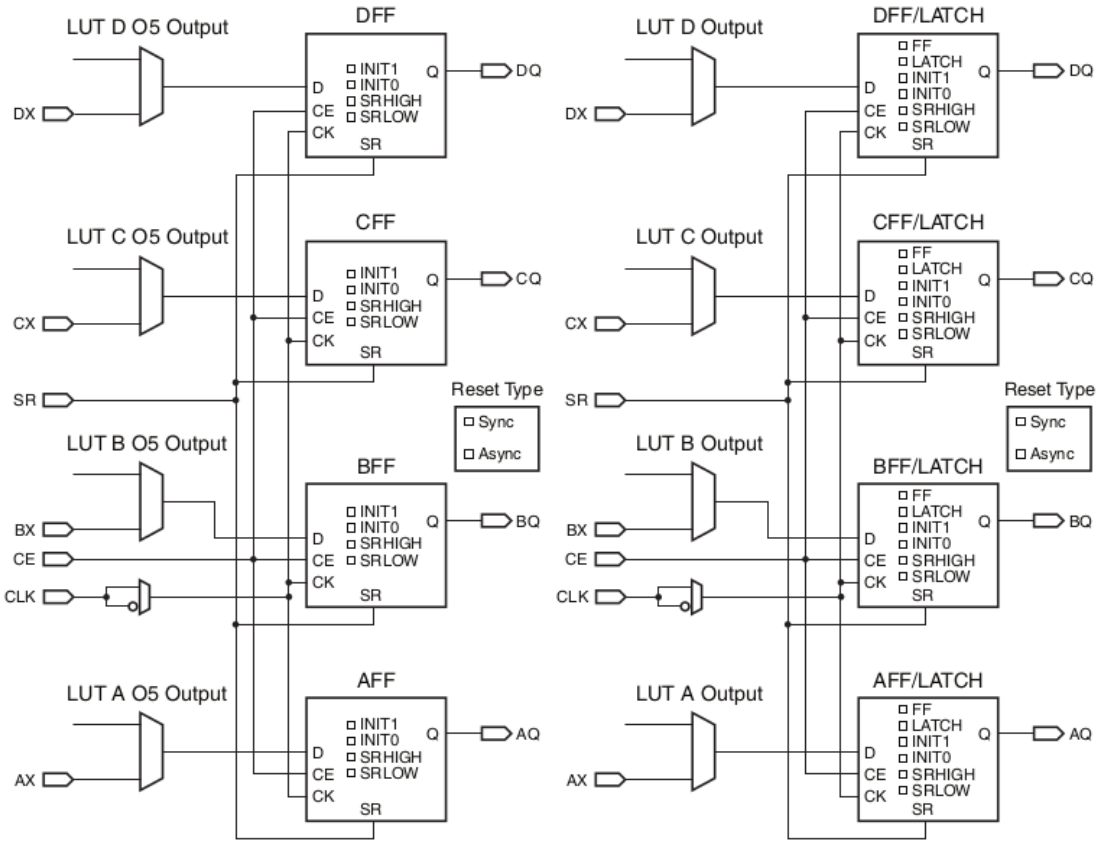
CLB

Architektura FPGA od společnosti Xilinx se skládá z několika základních prvků. Jedním je CLB (Configurable Logic Block). CLB je základním blokem pro vytváření logických funkcí. Každé CLB lze nakonfigurovat pro řešení požadované logické funkce. Na obrázku 2.1 je architektura CLB bloku. Každý blok obsahuje dva Slice bloky. Slice blok se skládá ze čtyř LUT (Look Up Table) a čtyř paměťových bloků. Slice bloky jsou v CLB na sobě nezávislé, avšak každý má propoj do Slice bloku sousedního CLB. CLB jsou uspořádány pod sebou ve sloupci a propoje mezi nimi jsou orientovány na jednu stranu. Tyto rychlé propoje se označují jako přenos a CLB bloky jsou takto propojeny pro realizaci rychlé aritmetiky. Přenos je v obrázku vyznačen jako CIN pro vstup z předchozího bloku a COUT jako výstup do dalšího. Normální vstupní a výstupní signály CLB jsou směrované přes propojovací síť (Switch Matrix).



Obrázek 2.1: Architektura CLB[6].

CLB obsahuje celkem osm LUT bloků. Každá LUT má šest vstupů a může být nakonfigurována tak, aby se chovala jako dvě LUT s pěti vstupy. Pak ale musí používat obě LUT stejné vstupy. Paměti mohou fungovat jako běžné registry, které reagují na nástupnou hranu hodin, nebo mohou být nakonfigurované jako registry typu latch. Na obrázku 2.2 je vidět zapojení paměťových modulů v jednom Slice bloku. Každý paměťový modul je připojen na výstup jedné LUT přes multiplexor, který může být nakonfigurovaný i na přeskočení LUT a přímé připojení jednoho ze vstupů A, B, C a D Slice bloku. Každá LUT má dva výstupy označené O6 a O5. O5 je použitý při konfiguraci s pěti vstupy. Každý výstup je připojen na jeden paměťový prvek. Na hlavním výstupu LUT (O6) je připojen prvek schopný fungovat jako klasický registr a také jako latch, na druhém výstupu (O5) je pouze klasický registr. Pokud je použita konfigurace na latch, registry na O5 výstupech použít nejdou. Mezi 25 až 50 procenty Slice bloků jsou SLICEM bloky. Ty mohou fungovat jako distribuovaná 64bitová RAM nebo jako posuvný registr. V případě konfigurace na posuvný registr jde o 32bitový nebo dva 16bitové registry. Ostatní bloky jsou označovány jako SLICEL.



Obrázek 2.2: Paměťové prvky Slice bloku[6].

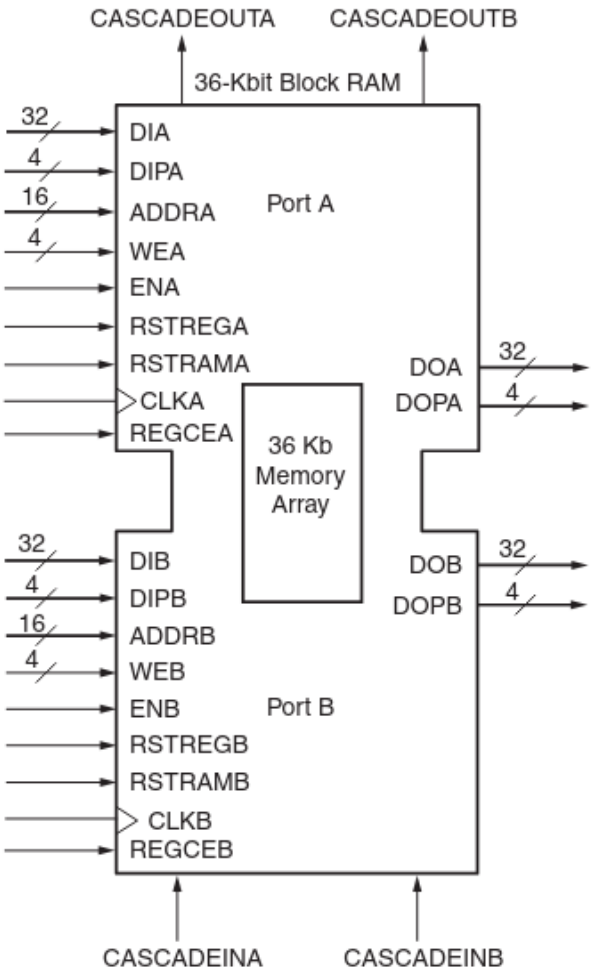
BlockRAM

Dalším prvkem programovatelné logiky jsou paměťové bloky BlockRAM, které slouží pro ukládání dat na čipu. Poskytují relativně velké množství paměti aniž by spotřebovávaly LUT pro registry. Paměťové bloky jsou přímo napojeny na programovatelnou logiku, přístup k nim je tedy velmi rychlý. Každý modul dokáže uložit 36KB dat a počet paměťových bloků se liší v závislosti na velikosti čipu. Malé čipy mají pouze 60 BlockRAM, zatímco velké čipy až 465. Šířka paměti je nastavitelná, paměti podporují několik různých šířek dat: 1, 2, 4, 8, 9, 16, 18, 32, 36, 64 a 72 bitů. Modul může být rozdělený na dvě plnohodnotné BlockRAM o poloviční velikosti. Rozdělené moduly ale mohou mít šířku dat pouze do 36 bitů. Dva nerozdělené moduly mohou být také spojeny v jednu velkou paměť o velikosti 64KB a šířce 1 byte. Paměti podporují detekci chyb a korekci. Dokáže provést korekci jednoho bitu a detekovat chybu ve dvou bitech. Pokud je paměť v ECC režimu, automaticky generuje bity Hammingova kódu při zápisu dat a provádí kontrolu při čtení.

BlockRAM má dva porty. Každý port je zcela nezávislý. Dají se použít nezávisle jak pro čtení, tak i pro zápis a stejně tak může být každý port nakonfigurován na libovolnou datovou šířku. Na obrázku 2.3 je schéma BlockRAM s dvojicí nezávislých portů A a B. Každý port má svojí adresu (ADDR), datový vstup (DI), datový výstup (DO), hodiny (CLK), povolení hodin (EN) a povolení zápisu (WE). Signály DIP a DOP jsou pro čtyři paritní bity, pokud parita není použita, můžou být bity použity pro data navíc. RSTREG je

synchronní set/reset výstupního registru, RSTRAM je pro případ, kdy je registr vypnutý v konfiguraci. Oba porty pracují synchronně s hodinami. Při náběžné hraně jsou vstupy uloženy do registrů a operace je provedena. V případě, že se jedná o zápis, je provedena za jeden hodinový cyklus. Operace čtení je provedena za dva hodinové cykly, pokud je použit výstupní registr. Pokud v konfiguraci paměti je výstupní registr vypnutý, data jsou vystavena na výstupu už po jednom hodinovém cyklu, avšak s prodlevou přístupu do RAM. S povoleným výstupním registrem lze operaci čtení zřetězit, a dosáhnout tak vyšší frekvence.

Při použití obou portů BlockRAM musí aplikace zařídit, aby nedošlo k zápisu na stejnou adresu ve stejný čas. Při takovém konfliktu dojde k nepředvídatelnému chování a zápisu nevalidních dat, avšak nemůže dojít k poškození modulu.



Obrázek 2.3: BlockRAM[7].

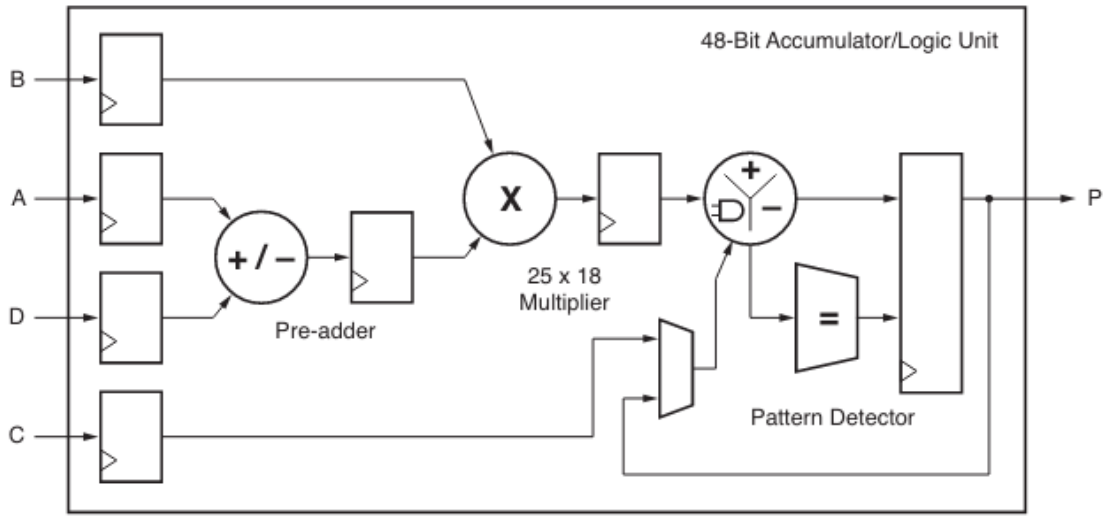
DSP

DSP (Digital Signal Processor) je dalším prvkem dostupným v programovatelné logice. Pro DSP u Zynqu použil Xilinx jejich variantu DSP48E1. Jeho funkcí je optimalizace aritmetických a logických výpočtů. Oproti implementaci výpočtu pomocí CLB, použitím DSP

prvku lze dosáhnout vyššího výkonu. Zároveň šetří místo na čipu, jelikož potřebné CLB by zabíraly více místa ve spoustě případů.

Na obrázku 2.4 je schéma základní funkčnosti DSP bloku. DSP obsahuje násobičku s operandy o šířce 25 a 18 bitů. Také obsahuje aritmeticko-logickou jednotku schopnou sčítat, odčítat a dokáže zpracovávat 10 logických funkcí. Mezi tyto logické funkce patří AND, OR, XOR a další. Další sčítačka (pre-adder) je předřazena ostatním operacím a poskytuje sčítání s nízkými energetickými nároky. Do aritmeticko-logické jednotky je vstup C multiplexovaný s výstupem. Zapojením výstupu na vstup DSP pracuje jako akumulátor, tento režim lze využít například k jednoduché implementaci čítače.

Detektor vzorů (pattern detector) může být nakonfigurován pro hledání různých situací, jako hledání přetečení nebo podtečení aritmetických operací, hledání středních bodů pro zaokrouhlování a dalších.



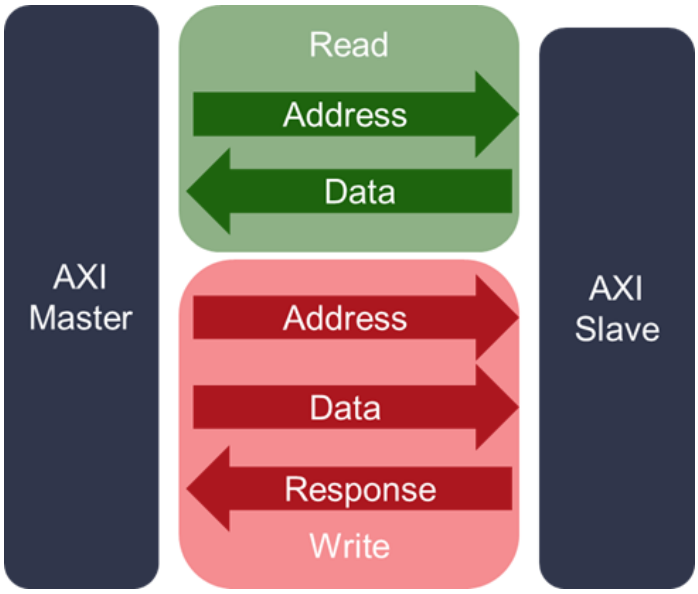
Obrázek 2.4: Schéma základní funkcionality DSP[5].

AXI

AXI je komunikační protokol pro přesun dat v hardwaru. AXI byla navržena především pro standardizovanou komunikaci mezi hardwarovými komponentami, zjednodušuje tak znovupoužitelnost komponent a umožňuje použití obecných komponent od dodavatelů třetí strany, čímž se snižuje cena vývoje. Platforma Zynq používá AXI sběrnice pro komunikaci programovatelné logiky s ostatními komponentami a je hlavním způsobem, který se používá. Sběrnice například umožňuje procesoru komunikovat s komponentou v FPGA nebo komponentě v FPGA přistupovat k paměti procesoru.

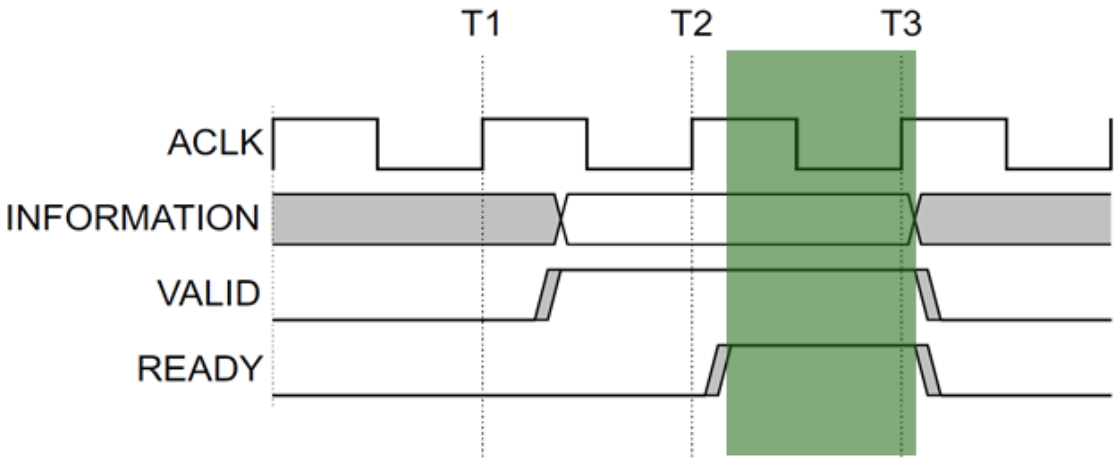
AXI protokol definuje komunikaci mezi dvěma komponentami na principu master a slave. Jedna komponenta je označena jako master a posílá požadavky pro čtení nebo zápis dat. Druhá komponenta je označená slave a požadavky vyřizuje. Na obrázku 2.5 je komunikace znázorněna. Na obrázku jsou dvě zařízení propojená AXI sběrnicí, u sběrnice je vyznačeno pět kanálů. Jako kanál se označuje každá datová linka, která kromě dat obsahuje další signály včetně signálů valid a ready. Na obrázku jsou vyznačené dva kanály potřebné pro čtení a tři pro zápis. Rozhraní pro čtení i zápis mají adresový a datový ka-

nál, zápisové rozhraní má navíc kanál pro odpověď. Protože jsou čtecí a zápisové rozhraní oddělené, komunikace je obousměrná, ale některé komponenty podporují pouze zápis, nebo čtení, potom může nepotřebné rozhraní chybět. Na sběrnici také může komunikovat mezi sebou více zařízení, k tomu se používají AXI přepínače, které rozdělují sběrnici a směrují komunikaci.



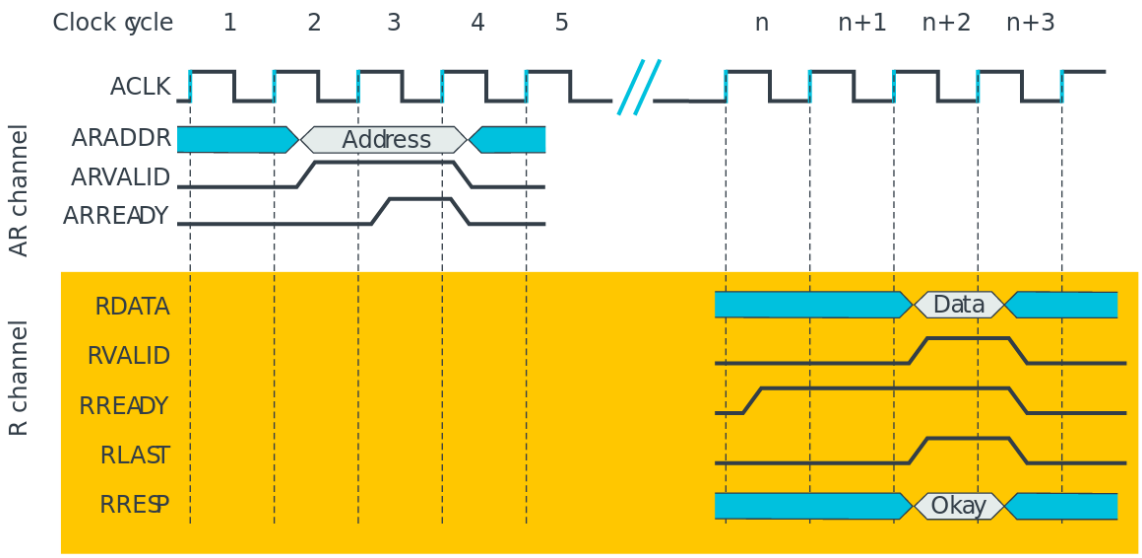
Obrázek 2.5: Schéma sběrnice AXI mezi dvěma komponentami[8].

Data se v kanálech přenášejí pomocí transferu, jenž nastává při náběžné hraně hodin, kdy jsou signály valid a ready v logické jedničce. Signál valid nastavuje odesílatel a signál ready nastavuje příjemce dat. Na obrázku 2.6 je zobrazen průběh přenosu dat. Na obrázku odesílatel nastaví data (information) a signál valid, indikující, že data jsou validní a transfer může začít. V čase označeném T2 na obrázku je signál valid aktivní, ale čeká se na signál ready, kterým příjemce signalizuje, že je připraven data přijímat. V čase označeném T3 jsou oba signály valid a ready aktivní a provede se transfer, v tu chvíli příjemce přečte data. Příjemce signál ready může nastavit kdykoli, i když není signál valid aktivní, transakce ale proběhne až je valid aktivní.



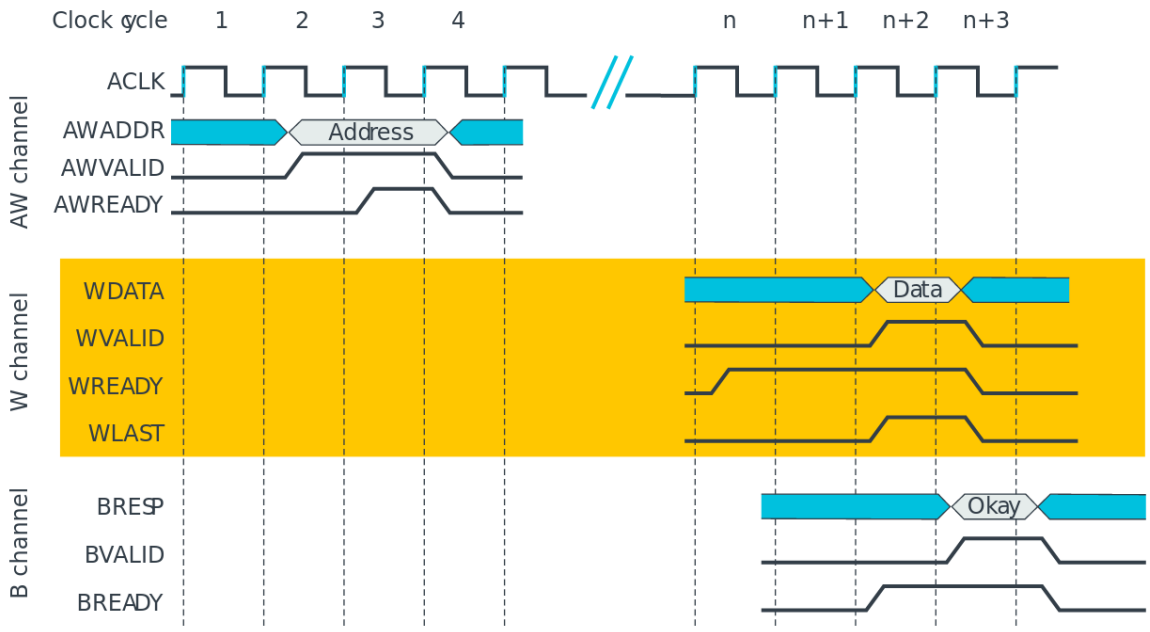
Obrázek 2.6: Transakce AXI protokolu[8].

Čtení dat probíhá pomocí dvou kanálů, kanálu pro zápis adresy a kanálu pro čtení dat. Na obrázku 2.7 je zobrazen průběh čtení pomocí AXI. Kanál pro adresu je na obrázku označen jako AR a kanál pro data je označený R. Čtení se skládá ze dvou transferů, prvním transferem se předá adresa na adresovém kanálu, kterou chce master přečíst. Druhým transferem slave předá master komponentě data podle adresy, kterou přijal. Na obrázku se provede první transfer s adresou na čtvrté náběžné hraně hodin a druhý transfer s daty proběhne při náběžné hraně označené jako n+2. Na obrázku jsou vyznačené další důležité signály, signál RLAST a PRESP. Signálem RLAST slave indikuje, že se jedná o poslední transfer, master totiž také může zažádat o více dat v dávkovém režimu. Signálem RRESP slave indikuje, zda došlo k úspěšnému čtení. Verze AXI4, která je použitá na platfomě Zynq, definuje určitá omezení. Při zápisu na sběrnici AXI4 musí proběhnout transfer s daty na další náběžné hraně hodin po předání adresy.



Obrázek 2.7: Čtení na sběrnici AXI[2].

Zápis dat probíhá pomocí tří kanálů a je zobrazen na obrázku 2.8. Kanál pro adresu má na obrázku označení AW, kanál pro data má označení W a kanál pro odpověď označení B. Zápis probíhá pomocí tří transferů, první trasfer je stejný jako u čtení, master předá adresu, na kterou chce zapisovat pomocí kanálu adresy. Druhým transferem master předá zapisovaná data kanálem pro data v čase n+1. Po zápisu slave předá informaci o úspěšnosti zápisu transferem na kanálu odpovědi (B), na obrázku je transfer na náběžné hraně n+3. Ve verzi AXI4 je při zápisu omezení, že slave musí provést transfer s odpovědí na zápis pomocí další náběžné hrany.

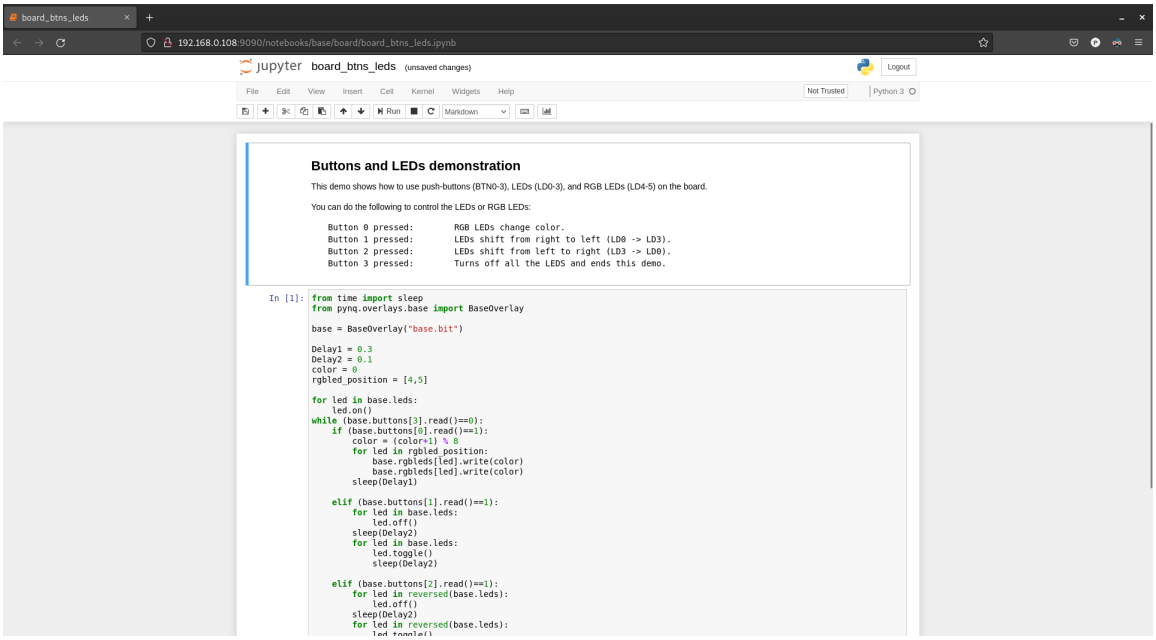


Obrázek 2.8: Zápis na sběrnici AXI[2].

Programovatelná logika má k dispozici několik AXI sběrnic v provedení master a slave. Sběrnice, které jsou označeny HP, jsou designované na maximální výkon. Pro zvýšení výkonu se HP sběrnice nedělí o přístup k paměti a obsahují také FIFO frontu.

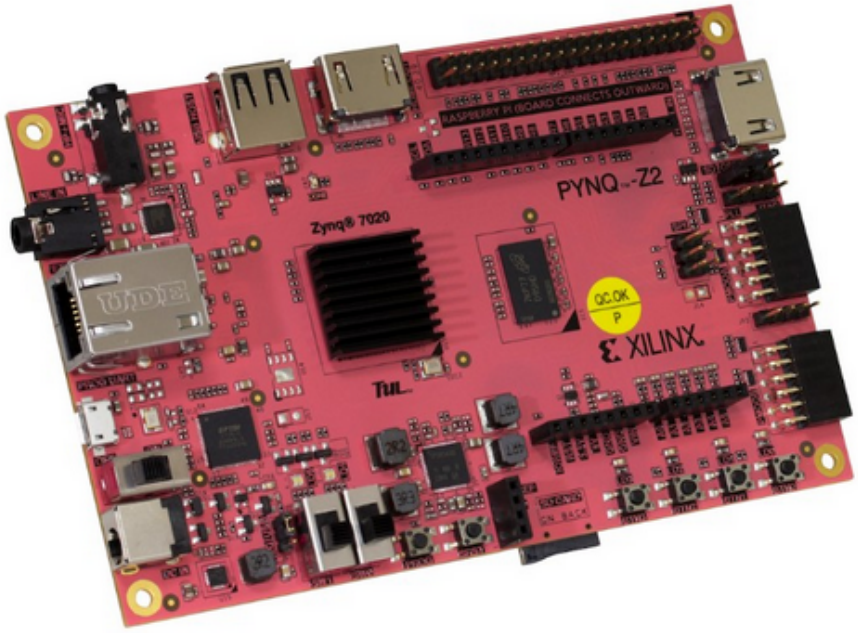
2.2 Zynq Pynq

Pynq je open source projekt ulehčující práci se Zynq platformou. Tato technologie umožňuje programovat Zynq procesor v pythonu. Python je jazyk s vysokou mírou abstrakce oproti C nebo C++, ve kterém se Zynq často programuje. Dělá tak platformu dostupnější studentům a umožňuje vývojářům být produktivnější. K dispozici je také python knihovna pro ovládání programovatelné logiky. V pythonu lze tedy implementovat kompletní aplikaci i s naprogramováním a komunikací s programovatelnou logikou. Pynq dává vývojářům přístup k ovládání platformy přes jednoduché webové rozhraní pomocí Jupyter notebooku. Jupyter notebook umožňuje psát a spouštět python kód přímo na Zynqu přímo přes webové rozhraní. To znamená, že platformu lze ovládat nezávisle na operačním systému a bez instalace vývojových nástrojů. Na obrázku 2.9 je Jupyter notebook v prohlížeči zachycený s otevřenou úlohou ze sady ukázkových úloh, které jsou dodávány v Pynq prostředí, jedná se o ukázku používání tlačítek a LED diod na přípravku. Na obrázku je vidět část Python kódu z úlohy, kde se programovatelná logika ovládá pomocí Pynq knihovny, která se importuje na druhém řádku.



Obrázek 2.9: Jupyter notebook.

Pynq podporuje řadu vývojových desek. Příkladem podporovaných desek jsou Pynq-Z1 a Pynq-Z2. Na obrázku 2.10 je deska Pynq-Z2 od výrobce TUL. Deska je osazena Zynq 7Z020 spárovaným s 512MB DDR3 RAM pamětí. Na desce je také slot pro SD kartu, ze které se standardně přípravek startuje. Na desce je gigabitový ethernet, přes který je také dostupný Jupyter notebook. Kromě rozhraní ethernet jsou na desce dostupné i další rozhraní jako USB 2.0, HDMI nebo audio jack. Samozřejmě jsou na desce vyvedené i i/o (vstupně-výstupní) piny čipu a to na dva standardní Pmod porty a standardní kolíkové a dutinkové lišty. Lišty jsou na desce umístěné tak, aby byly kompatibilní s Arduino shield deskami a Raspberry Pi konektory. A jako na většině vývojových desek zde nechybí ani několik tlačítek a LED diod.

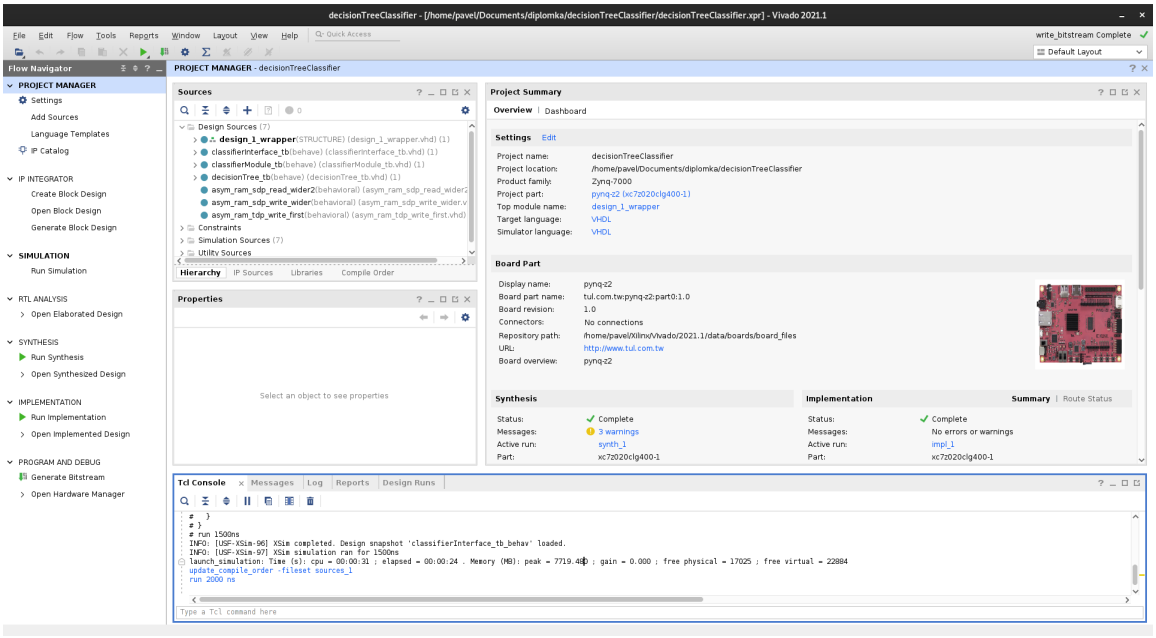


Obrázek 2.10: Pynq Z2[9].

2.3 Popis vývojových nástrojů

Pro vývoj akcelerátorů na platformu Pynq se používá vývojové prostředí Vivado. Vivado je vývojové prostředí vyvinuté firmou Xilinx a umožňuje vývoj hardware, včetně syntézy a testování. Vivado umožňuje vytvářet akcelerátory pro FPGA od firmy Xilinx. Jelikož je platforma Pynq založena na platformě od firmy Xilinx, používá se pro vývoj.

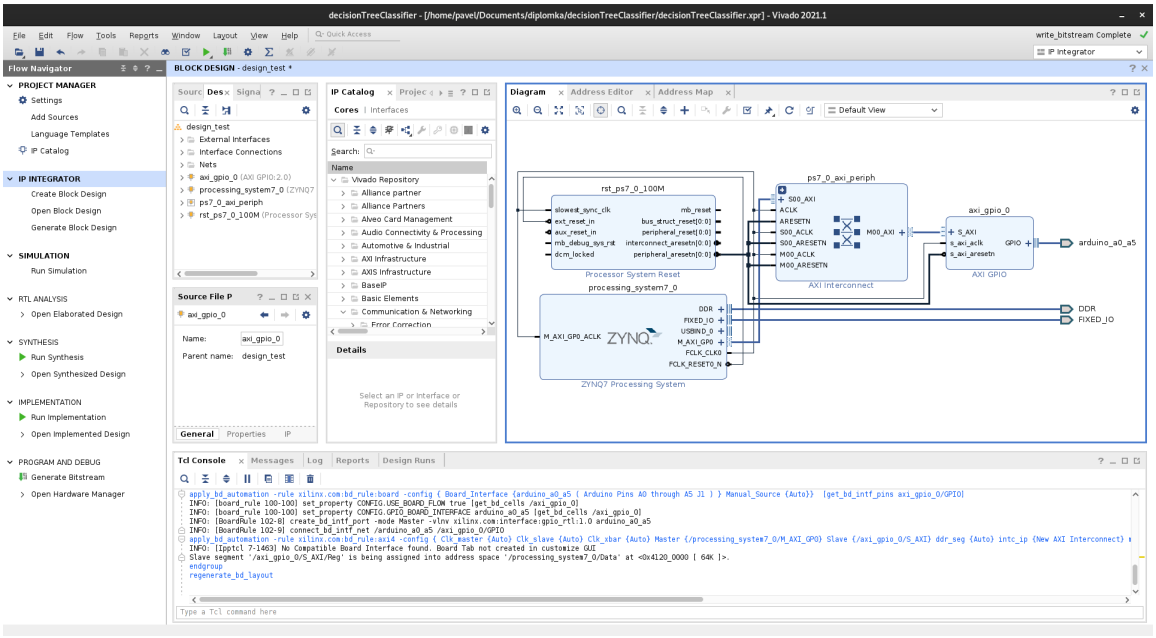
Na obrázku 2.11 je snímek vývojového prostředí Vivado s otevřeným projektem pro platformu Pynq. Uprostřed je seznam zdrojových souborů projektu, který umožňuje soubory spravovat a editovat. Napravo je přehled projektu, zde se dá najít spousta informací o projektu, jako je například počet spotřebovaných zdrojů na FPGA nebo odhadovaná spotřeba. Na dolní straně je vidět Tcl konzole. Tcl jazykem lze Vivado řídit, ačkoli běžné úkony zvládne uživatelské rozhraní. A na levé straně je navigátor poskytující rychlý přístup k často používaným operacím.



Obrázek 2.11: Vývojové prostředí Vivado.

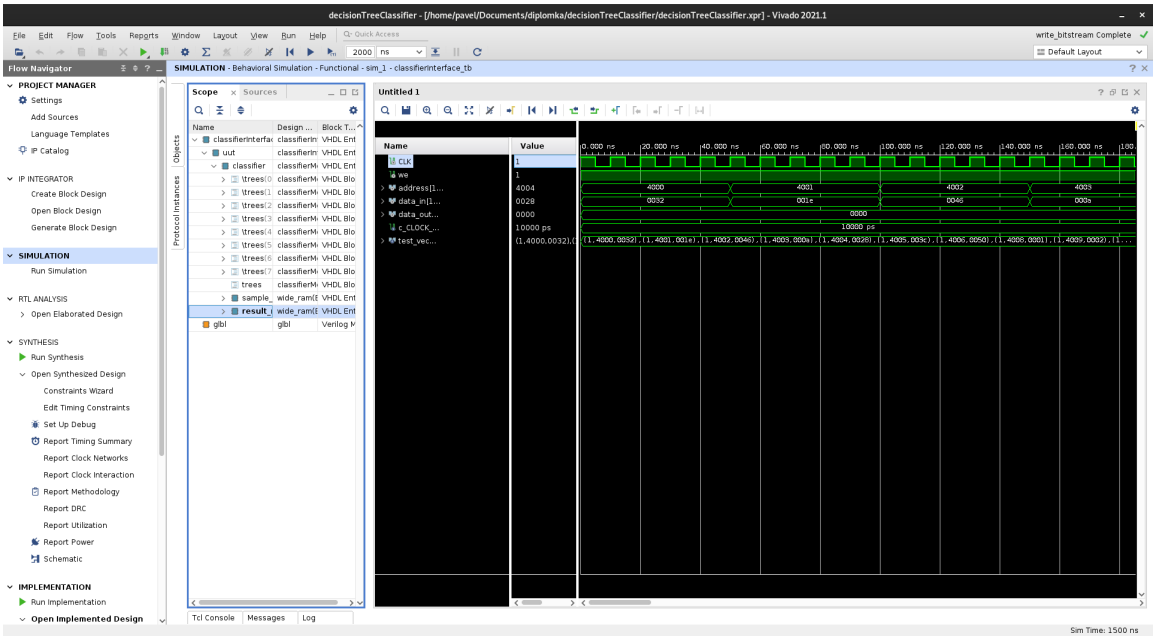
Vivado má vestavěný editor zdrojových souborů, který obsahuje běžné nástroje jako kontrolu syntaxe, našeptávač a další. Podporuje jazyky VHDL, Verilog a SystemVerilog. Do Vivado balíku patří také Vivado HLS, které provádí vysokoúrovňovou syntézu, a vytvořené moduly lze jednoduše integrovat do projektu ve Vivado. Vysokoúrovňová syntéza podporuje jazyky C, C++, SystemC a OpenCL.

Vivado obsahuje blokové prostředí, které umožňuje vytvářet blokové designy. Jedná se o grafický editor. Do prostředí lze vkládat vlastní moduly napsané v jednom z podporovaných jazyků ve formě bloků a bloky mezi sebou propojovat. Kromě vlastních modulů lze používat předem připravené bloky z knihovny nebo importovat z externího zdroje. Pokud má implementace bloku nastavitelné parametry, lze je v blokovém prostředí editovat dvojitým kliknutím na příslušný blok. V prostředí lze kromě bloků definovat také vstupní a výstupní porty, které lze navázat na piny FPGA. Blokové prostředí je zobrazeno na obrázku 2.12. Na obrázku je na pravé straně blokový editor se čtyřmi propojenými bloky a několika porty. Zobrazené zapojení vytvoří akcelerátor na platformu Pynq, který umožní aplikaci na procesoru ovládat množinu pinů na přípravku jako vstupně-výstupní piny.



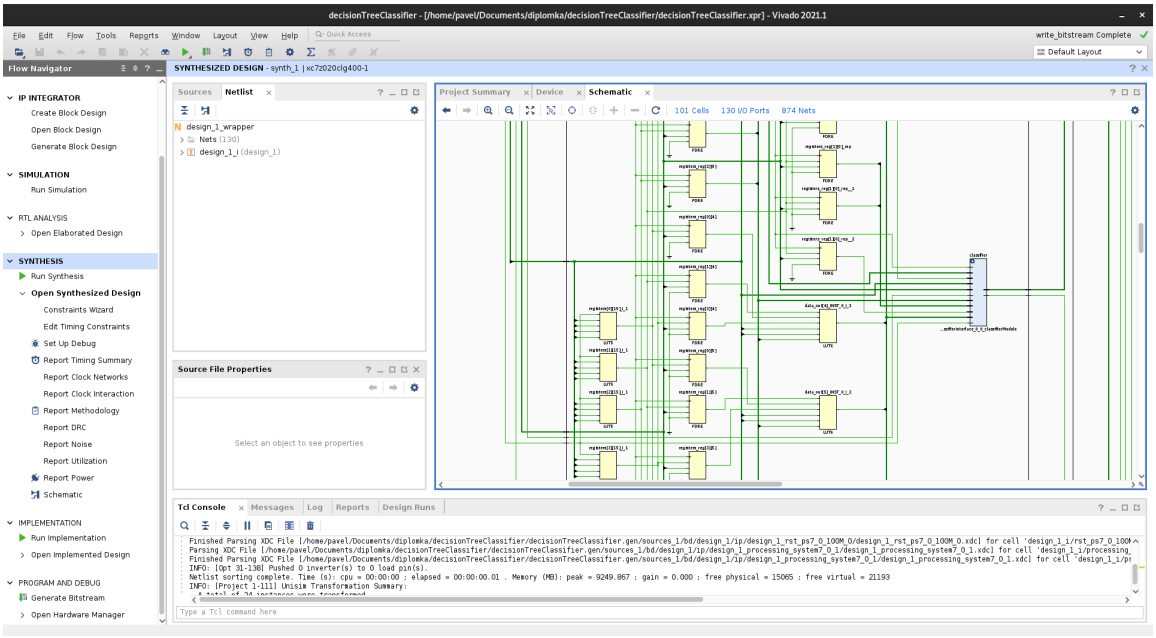
Obrázek 2.12: Blokové prostředí Vivado.

Pro otestování implementace je ve Vivadu možnost využít vestavěný simulátor. Na obrázku 2.13 je zobrazen výstup ze simulátoru ve Vivadu. Na pravé straně je zobrazen průběh signálů v čase. Na levé straně je průzkumník simulovaných modulů. V průzkumníkovy lze nalézt v hierarchické struktuře všechny moduly v simulovaném akcelerátoru a prohlížet hodnoty jejich vnitřních signálů. Také lze z průzkumníka přidat libovolný signál do grafu průběhu signálů aby šlo sledovat jeho hodnoty v čase. Pro zobrazení průběhu hodnot nově přidaného signálu musí být ale simulace spuštěna znovu.



Obrázek 2.13: Simulátor ve Vivado.

Schémata vytvořených akcelerátorů lze ve Vivadu prohlížet v grafickém prohlížeči. Na obrázku 2.14 je grafický prohlížeč výsledku syntézy. Na obrázku jsou vidět využité LUT tabulky, registry a také moduly. Jeden modul je vidět v obrázku, je vyznačen modře. Moduly zvyšují přehlednost schématu, protože odpovídají jednotlivým instancím modulů v kódu, například instancím architektur ve VHDL. Moduly lze libovolně rozbalit a zobrazit jejich obsah. U každé LUT tabulky lze zobrazit i změnit její pravdivostní tabulku.

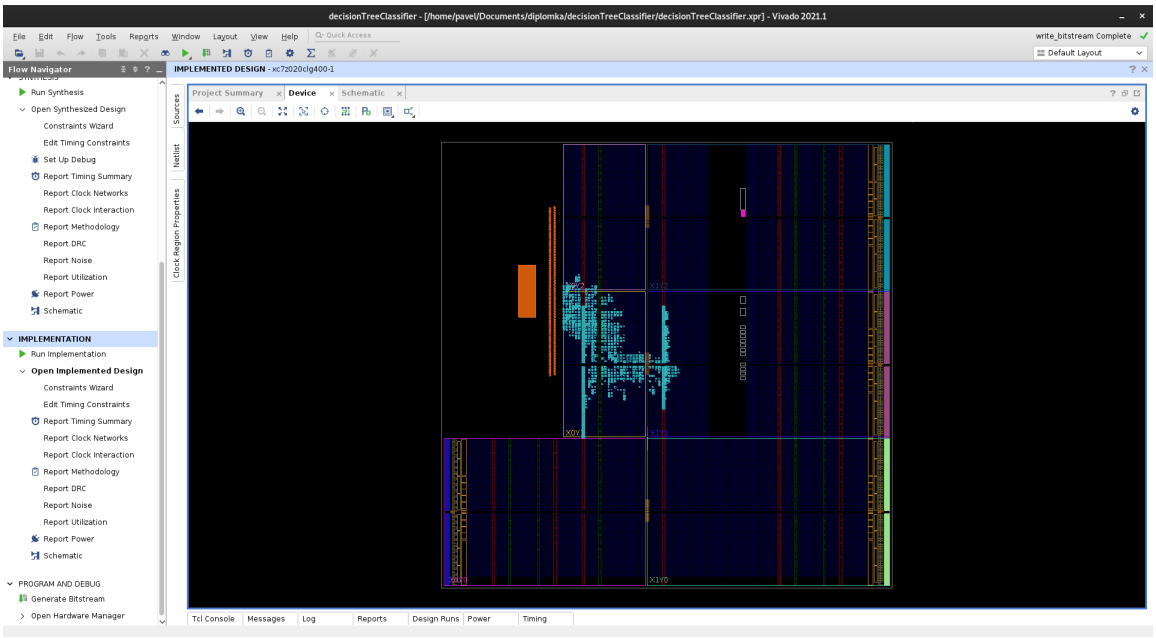


Obrázek 2.14: Vizuální prohlížeč výsledku syntézy ve Vivado.

Pro vývoj na specifickou platformu se ve Vivado projektu musí vybrat specifické FPGA nebo lze vybrat vývojovou desku, která obsahuje všechny potřebné informace pro Vivado. Některé vývojové desky jsou dostupné ve Vivadu v základu, včetně některých desek podporovaných platformou Pynq. Z předinstalovaných Pynq desek je ve Vivadu k dispozici například deska ZCU104, ale ostatní desky, jako je Pynq Z2, na kterou se tato práce soustředí, se musí doinstalovat manuálně. Pro instalaci vývojových desek se musí stáhnout archív, většinou je dostupný od výrobce, a extrahovat soubory do složky *<instalační složka Xilinx>/Vivado/<verze>/data/boards/board_files*. Poté je vývojová deska ve Vivado dostupná a lze vytvořit projekt. Po vytvoření projektu cíleného na vývojovou desku Pynq, je vhodné vytvořit blokové schéma a nastavit ho jako výchozí modul (top modul). Blokové schéma však po vytvoření nelze nastavit jako výchozí modul, musí se vytvořit obálka (wrapper) v cílovém jazyku, který je nastaven v projektu. Obálka lze ve Vivadu automaticky vygenerovat pravým kliknutím na blokové schéma v seznamu zdrojových souborů a zvolením možnosti "Create HDL Wrapper". Dále je vhodné vložit do blokového diagramu blok "Zynq 7 Processing system", který funguje jako rozhraní pro komunikaci s procesorem na čipu. A poté spustit automatizaci zapojení, tlačítko se po vložení bloku zobrazí na horním okraji editoru. Automatizace vygeneruje zapojení DDR paměti a základních pinů.

Proces převodu zdrojových souborů do výsledného binárního souboru, který funguje k nakonfigurování vytvořeného akcelerátoru do FPGA, je ve Vivadu rozdělen do tří částí. Všechny tři části lze spustit z navigátoru vlevo, jsou jimi syntéza, implementace a generování binárního souboru (generate bitstream). Syntéza ze zdrojových souborů vytváří obvod z obecných hardwarových bloků. Implementace mapuje výstup syntézy do konkrétní technologie a generování binárního souboru vytvoří binární konfigurační soubor pro konkrétní FPGA. Po spuštění implementace lze ve Vivadu zobrazit vytvořený akcelerátor namapovaný na konkrétní prvky FPGA, na obrázku 2.15 jsou zobrazeny využité prvky při mapování akcelerátoru na technologii Zynq. Na obrázku je v prohlížeči zobrazen konkrétní čip se všemi jeho prvky, je zde vidět rozdělení čipu na šest hodinových zón a v levém horním rohu, kde

je prázdné místo, se na čipu nalézá procesor. Použité prvky jsou na obrázku v prohlížeči zvýrazněné světle modře, při implementaci většina zdrojů byla v tomto případě použita ze zóny X0Y1, pravděpodobně proto, že akcelerátor je napojen na AXI sběrnici vyvedenou z procesoru a algoritmus implementace využil nejbližší zdroje, aby minimalizoval vzniklé zpoždění v propojovací síti.



Obrázek 2.15: Vizuální prohlížeč mapování na architekturu ve Vivado.

2.4 Hledání řetězců

Hledání řetězců je v informatice velmi základní operace. Jde o vyhledávání výskytů hledaného řetězce v jiném, zpravidla výrazně delším, řetězci. Úkolem je najít všechny výskyty hledaného řetězce, někdy se ale hledá pouze první výskyt. Hledání řetězců se provádí v každém odvětví informatiky a množství prohledávaných dat je velké a stále se zvětšuje. Velká data se prohledávají například v bioinformatice, kde se vyhledává v proteinových a DNA sekvencích, nebo internetové bezpečnosti, kde se vyhledávají hrozby v síťovém provozu. Pro hledání řetězců existují různé algoritmy s různými vlastnostmi. Algoritmy se většinou poměřují nejhorším a průměrným časem výpočtu.

Nejjednodušší algoritmus prohledává text od začátku a posunuje se znak po znaku. Na každé pozici v textu se pokaždé porovnávají znaky od začátku hledaného řetězce, dokud se rovnají. Pokud se všechny znaky hledaného řetězce rovnají, řetězec byl nalezen. Pokud se znaky nerovnají, přejde se na další pozici v textu. Tomuto algoritmu se někdy říká algoritmus hrubé síly nebo naivní algoritmus. Jeho časová složitost je $O(mn)$, kde m je délka textu a n je délka vyhledávaného řetězce.

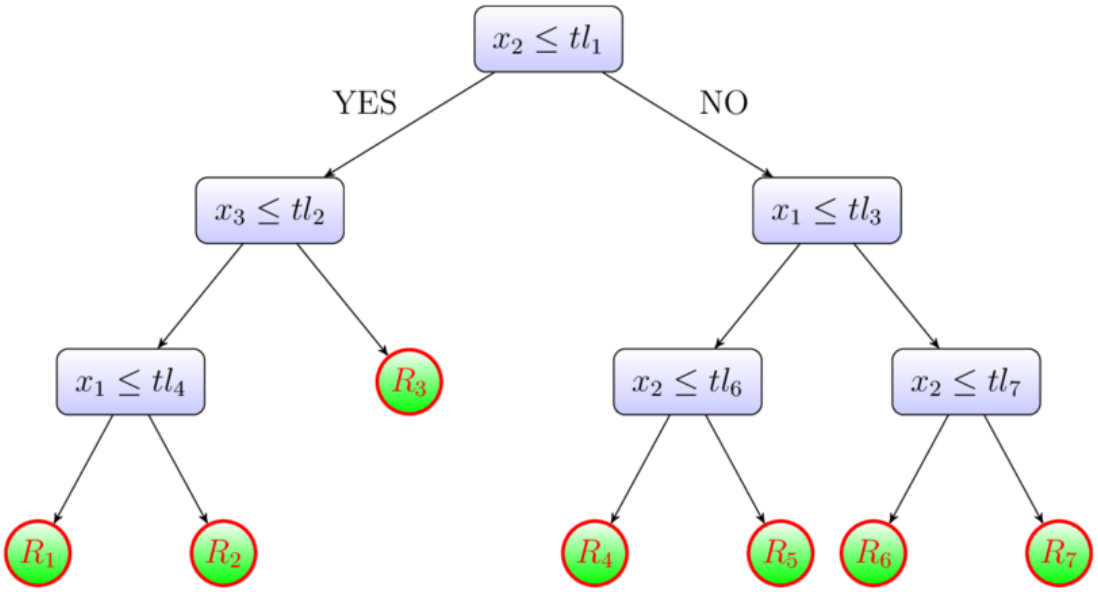
Komplexnějším algoritmem je algoritmus Knuth–Morris–Pratt, také označovaný zkratkou KMP. Algoritmus oproti naivnímu algoritmu počítá se znalostí předchozích porovnání. Algoritmus se nevrací k předchozím znakům v prohledávaném textu, pokud porovnání selže, ale je schopen detekovat zda v textu přečetl začátek hledaného řetězce (prefix) a pokračovat

s vyhledáváním s danou informací. KMP funguje tak, že před vyhledáváním si vytvoří pomocné pole o délce hledaného řetězce, kde každ7 prvek odpovídá znaku v hledaném řetězci a má hodnotu počtu znaků prefixu, který byl přečten ve chvíli, kdy dojde k porovnání se znakem na stejné pozici. Stav výpočtu se skládá ze dvou hodnot, pozice v textu a pozice v hledaném řetězci. Při výpočtu se obě pozice zvyšují a znaky na těchto pozicích se porovnávají. Když se porovnání nepovede, pozice v hledaném řetězci se nastaví na hodnotu z pomocného pole ze stejné pozice, jako je aktuální pozice v hledaném řetězci. KMP algoritmus má časovou složitost $(m + n)$, kde m je délka textu a n je délka vyhledávaného řetězce.

2.5 Rozhodovací stromy

Rozhodovací strom je pojem ze strojového učení. Rozhodovací strom je model, se kterým pracují různé algoritmy strojového učení. Model rozhodovacího stromu se pomocí zvoleného algoritmu natrénuje na datech. Poté dokáže klasifikovat nová data. Jedná se dnes o jeden z jednodušších přístupů, jiné modely jako neuronové sítě zpravidla dosahují lepších výsledků. Rozhodovací stromy však umožňují jednoduché prozkoumání natrénovaného modelu, protože je strom dobře pochopitelný a lze ho přehledně vizualizovat.

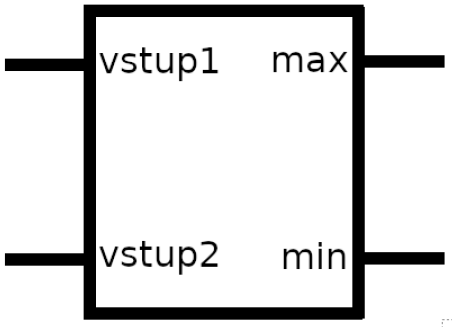
Rozhodovací stromy jsou stromy, některé algoritmy vytvářejí pouze binární stromy, ale obecně mohou mít potomků více. V uzlech rozhodovacích stromů jsou pravidla, podle kterých se data klasifikují. A v listových uzlech jsou výsledné třídy, do kterých se data klasifikují. Na obrázku 2.16 je příklad jednoho rozhodovacího stromu. Klasifikovaným datům se také někdy říká vzorky, vzorky mohou mít různé množství atributů podle úlohy. Na vzorky se vždy aplikuje pravidlo z jednoho uzlu stromu. Začíná se s kořenovým uzlem a podle výsledku pravidla se určí následný uzel. Strom na obrázku je binární a obsahuje pravidla, která porovnávají atributy vzorku x1 až x3 s konstantami tl1 až tl7. Pokud je výsledek porovnání kladný, pokračuje se levým potomkem, v opačném případě pravým. Jakmile se narazí na listový uzel, výsledek je jeho hodnota. Na obrázku jsou třídy, do kterých se vzorky rozdělují R1 až R7.



Obrázek 2.16: Rozhodovací strom[1].

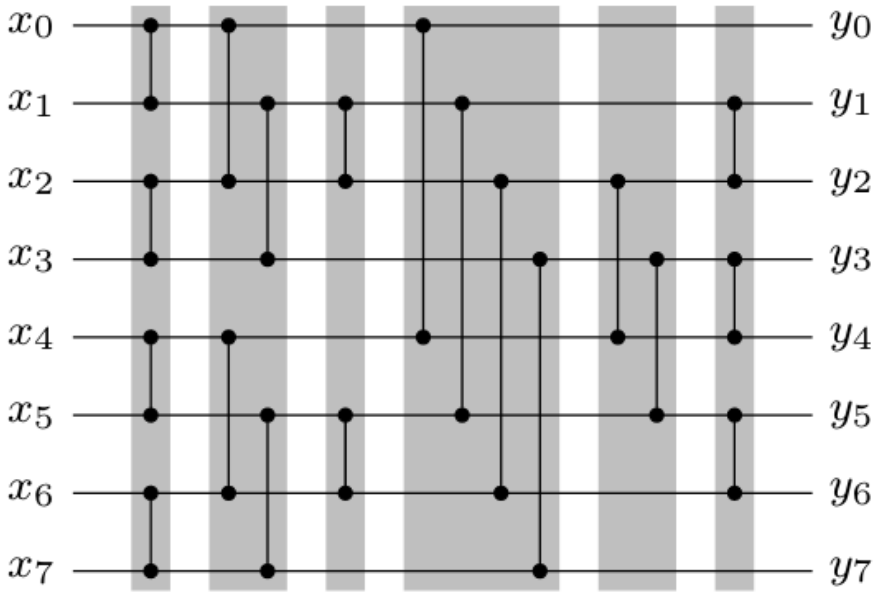
2.6 Řadicí sítě

Řadicí sítě se používají pro řazení dat. Síť se skládá z porovnávacích bloků, schéma porovnávacího bloku je na obrázku **??**. Porovnávací blok je dvouvstupová komponenta s dvěma výstupy. Hodnoty vstupu bloku se mezi sebou porovnají a na výstup max se vloží větší a na výstup min menší z porovnávaných hodnot. Zapojením těchto bloků do sebe lze vytvořit řadicí síť o požadovaném počtu vstupů. Schéma jedné řadicí sítě je na obrázku 2.18. Na levé straně jsou označené vstupy jako x0 až x7 a výstupy jsou na pravé straně označené y0 až y7. Vertikální čáry propojují vstupy, které jsou připojené na porovnávací blok.



Obrázek 2.17: Porovnávací blok.

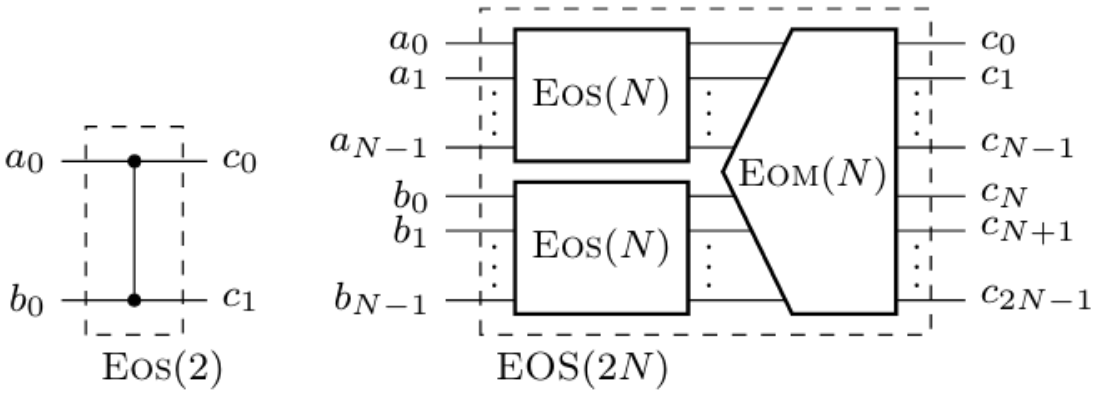
Řadicí sítě lze jednoduše paralelizovat. Pro paralelizaci stačí výpočet zřetězit rozdělením výpočtu po jednotlivých úrovních porovnávacích bloků. V úrovních jsou vždy bloky, které se neovlivňují, tedy jejich výstupy nejsou zapojené do žádného ze vstupů ostatních bloků. Na obrázku 2.18 jsou nezávislé bloky vyznačeny šedými pruhy. Protože jsou řadicí sítě dobře



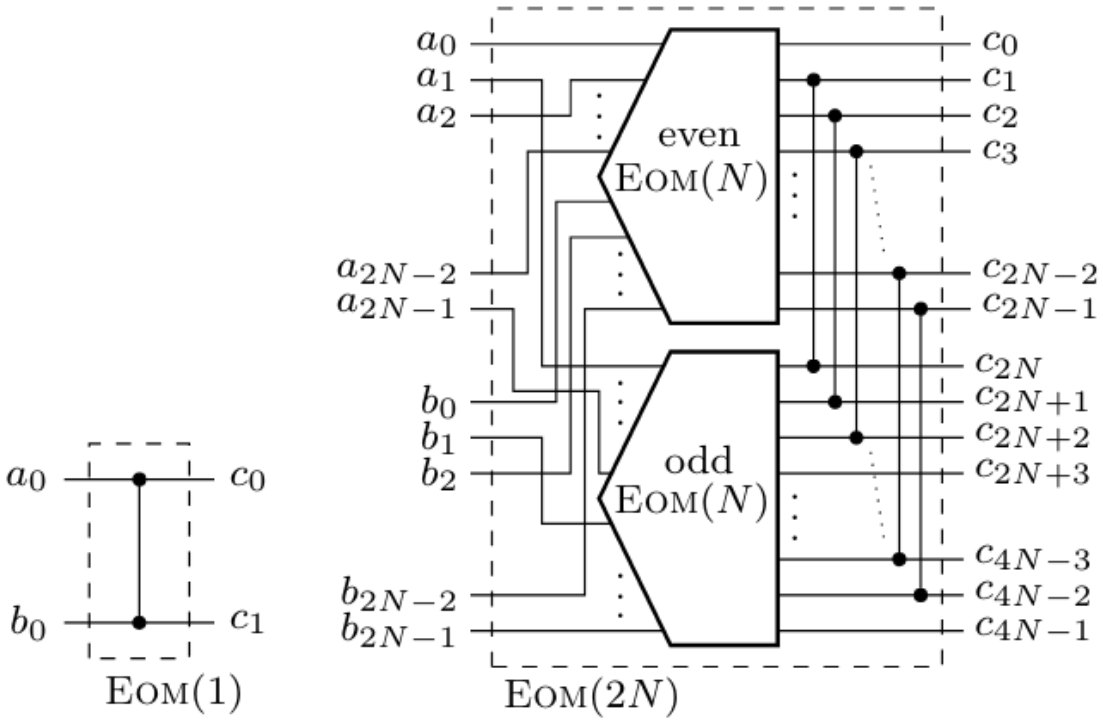
Obrázek 2.18: Řadicí síť[3].

paralelizovatelné, jsou vhodné pro implementaci v hardwaru. Řadicí sítě nejsou vhodné pro velké množství vstupních dat, protože jejich velikost roste s velikostí vstupů. Také nejsou vhodné, pokud velikost vstupu není předem známá.

Pro vytvoření řadicí sítě existují různé algoritmy, jedním z nich je Even-odd Merging. Síť na obrázku 2.18 byla vytvořena tímto algoritmem. Even-odd Merging má rekurzivní definici, na obrázku 2.19 je definice řadicí sítě označená jako Eos. Na obrázku je vlevo řadicí síť pro dva elementy, která se skládá pouze z jednoho porovnávacího bloku, a vpravo je řadicí síť pro 2N vstupů. Síť pro 2N vstupů seřadí každou polovinu vstupů pomocí řadicí sítě pro N vstupů a poté je sloučí do jedné seřazené sekvence pomocí bloku Eom. Blok Eom je také definovaný rekurzivně. Na obrázku 2.20 je nalevo definice pro eom, který sloučí dvě sekvence o velikosti jedna, a napravo pro velikosti 2N. [3]



Obrázek 2.19: Rekurzivní definice řadicí sítě Even-odd Merging[3].



Obrázek 2.20: Rekurzivní definice bloku Eom[3].

Kapitola 3

Shrnutí požadavků na návrh úloh

Cílem práce je vytvořit sadu ukázkových úloh na podporu výuky. Práce zahrnuje tři úlohy demonstrující hardwarovou akceleraci na přípravku Pynq Z2. V souladu se zadáním bude první úloha zaměřena na akceleraci hledání řetězců. Druhá úloha se bude týkat práce s maticemi a třetí úloha bude z oblasti klasifikace dat pomocí rozhodovacích stromů. Všechny úlohy budou navrženy tak, aby šlo pomocí parametrů nastavit míru paralelního zpracování (zrychlení) a množství zabraných hardwarových zdrojů. Úlohy jsou detailně navrženy v kapitole 4. Po implementaci úloh se ověří jejich funkčnost v simulacích a na reálném hardware. Zhodnotí se míra dosaženého zrychlení úlohy oproti řešení na procesoru. A zhodnotí se množství využitých hardwarových zdrojů.

Obecné požadavky na všechny tři úlohy jsou stejné. Jelikož příklady mají fungovat jako ukázkové příklady, měly by být úlohy jednoduché k použití a vyzkoušení. Primárním cílem není, aby byly prakticky použitelné v reálných aplikacích, důležitější je snadná pochopitelnost a přehlednost kódu. Pro popis hardwaru je tak nutné použít pouze čistý VHDL kód. Jak bylo řečeno dříve, všechny úlohy bude možné parametrizovat. Tyto parametry budou zadávané v době návrhupomocí VHDL parametrů. Všechny tyto parametry by měly být v jednom VHDL souboru i s dokumentací jejich funkce pro přehlednost.

V úloze na hledání řetězců bude možné zvolit maximální počet a délku hledaných řetězců. Další parametr bude udávat míru paralelního zpracování, a tím ovlivňovat rychlost vyhledávání. Počet spotřebovaných zdrojů budou přirozeně ovlivňovat všechny tyto parametry. V reálné aplikaci by data mohla přicházet po nějaké sériové lince,zaměření této úlohy je ale na urychlení samotného výpočtu a ne předávání dat. Pro jednoduchost použití bude data předávat aplikace pracující na procesoru. Text pro vyhledávání bude po předání programovatelné logice uložen do BlockRAM modulů, z kterých se budou po spuštění data číst podobně jako v případě, kdy data přichází z rychlého vstupního rozhraní. Protože cílem této aplikace není dosáhnout nejvyšší možné rychlosti, ale demonstrovat hardwarovou akceleraci, úloha bude implementovat pouze naivní vyhledávací algoritmus, který porovnává vstupní proud dat znak po znaku. A ve zhodnocení bude také porovnávána se stejným algoritmem na procesoru.

Kapitola 4

Návrh úloh

V této kapitole jsou specifikovány všechny tři vytvářené úlohy. Je zde popsáno, jak byla úloha navržena s ohledem na požadavky z kapitoly 3. Následně je uveden návrh architektury a implementace aplikace. Všechny návrhy byly provedeny po konzultaci s vedoucím práce.

4.1 Systém pro transport dat do FPGA

Systém pro transport dat do FPGA zajišťuje komunikaci s obslužnou aplikací, která funguje na procesoru, s výpočetním prvkem akcelerátoru. Tento systém byl navržen tak, aby byl znovupoužitelný ve všech úlohách s minimálními změnami. Úlohy se v této práci zaměřují převážně na akceleraci samotného výpočtu a v kapitole 3 bylo odůvodněno, proč na transport dat do FPGA se úloha nezaměřuje. Přestože transport dat není primárním cílem úloh, byla potřeba vytvořit nějaký systém pro předávání vstupních dat, přijímání výstupních dat a ovládání akcelerátoru. Z toho důvodu systém nebyl navrhován na dosažení maximální rychlosti, ale byl navržen pro jednoduchost systému a zlepšení čitelnosti a pochopitelnosti kódu.

Systém řeší dvě důležité funkce, z nichž první je komunikace s procesorem a druhá spočívá v komunikaci s jinými částmi akcelerátoru. Komunikace s procesorem zahrnuje předávání vstupních dat akcelerátoru a předávání výsledků zpět procesoru. Kromě předávání těchto dat systém zajišťuje předávání řídicích signálů, jako je spuštění výpočtu, a dalších konfigurací. Každá úloha je však jiná a potřebuje předávat jiné data, proto byla snaha navrhnout systém více obecně.

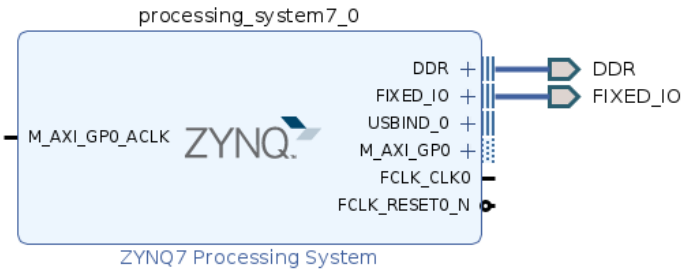
Druhá funkce systému spočívá v zajištění komunikace s jinými částmi akcelerátoru. Procesor komunikuje se systémem, který pak dále zajišťuje zpracování a předávání těchto dat správným komponentám akcelerátoru. Řídicí signály jsou uloženy v paměti systému a bez prodlení zpřístupněny konkrétním komponentám. Konfigurační příkazy systém pouze předává konkrétním komponentám, oproti řídicím signálům je ale neukládá a nechává jejich zpracování zcela na konfigurované komponentě. Nejsložitější zpracování je však při práci se vstupními a výstupními daty. Například u úlohy "Klasifikace pomocí rozhodovacích stromů"jsou vstupní data klasifikované vzorky a výstupní data jsou výsledné třídy. Jak bylo řečeno v kapitole 3, tato data jsou uložena v paměti akcelerátoru, tento systém spravuje tuto paměť a poskytuje rozhraní pro přístup k paměti ostatním komponentám.

4.1.1 Komunikace s procesorem

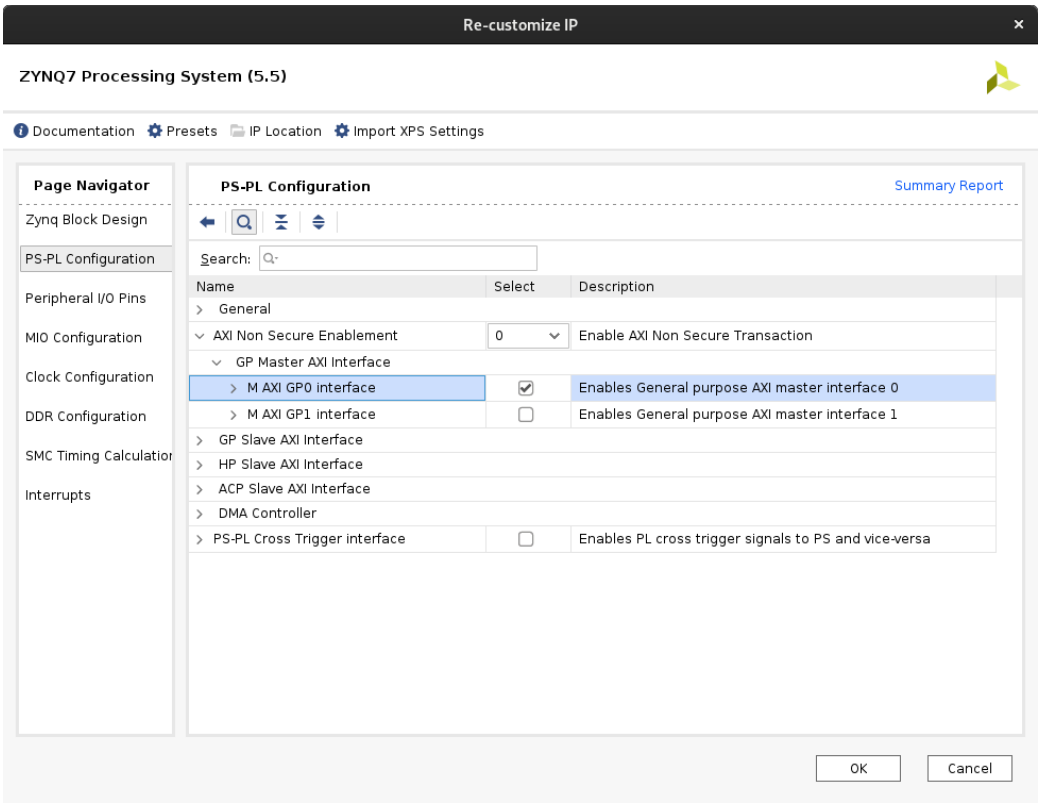
Pro komunikaci hardwaru s procesorem používá platforma Zynq sběrnice AXI. AXI má relativně jednoduchý protokol a nebylo by příliš těžké vytvořit komunikátor používající AXI, přesto se v práci zvolilo nekomunikovat přes sběrnici přímo, ale použít AXI GPIO IP blok, který je dodávaný s Vivado softwarem. GPIO blok byl zvolen kvůli tomu, že by měl být pochopitelnější pro studenty oproti AXI, jelikož jeho chování je principiálně jednodušší, a také kvůli tomu, že studenti už mohou být seznámeni s konceptem GPIO (univerzální vstup/výstup). Použití tohoto bloku způsobí pomalejší komunikaci oproti používání AXI sběrnice napřímo, ale jak bylo dříve zdůvodněno, jednoduchost řešení je důležitější.

Zvolený GPIO blok je dostupný jako součást software Vivado a lze jednoduše zapojit v blokovém prostředí (blockdesign). Blok se připojuje k AXI sběrnici a komunikuje s procesorem jako slave. GPIO blok procesoru umožňuje nastavovat logické úrovně na jeho výstupy a číst jeho vstupy. Blok má dva GPIO kanály a každý kanál má šířku 32 bitů. Každý bit kanálu má jeden výstupní a jeden vstupní signál z bloku. Také je vyveden jeden signál signalizující, zda je bit momentálně přepnut jako vstupní, nebo výstupní. Pro použití druhého kanálu se ale musí v konfiguraci bloku kanál aktivovat. U každého kanálu lze také nakonfigurovat jeho šířka až do jeho maximální šířky třiceti dvou.

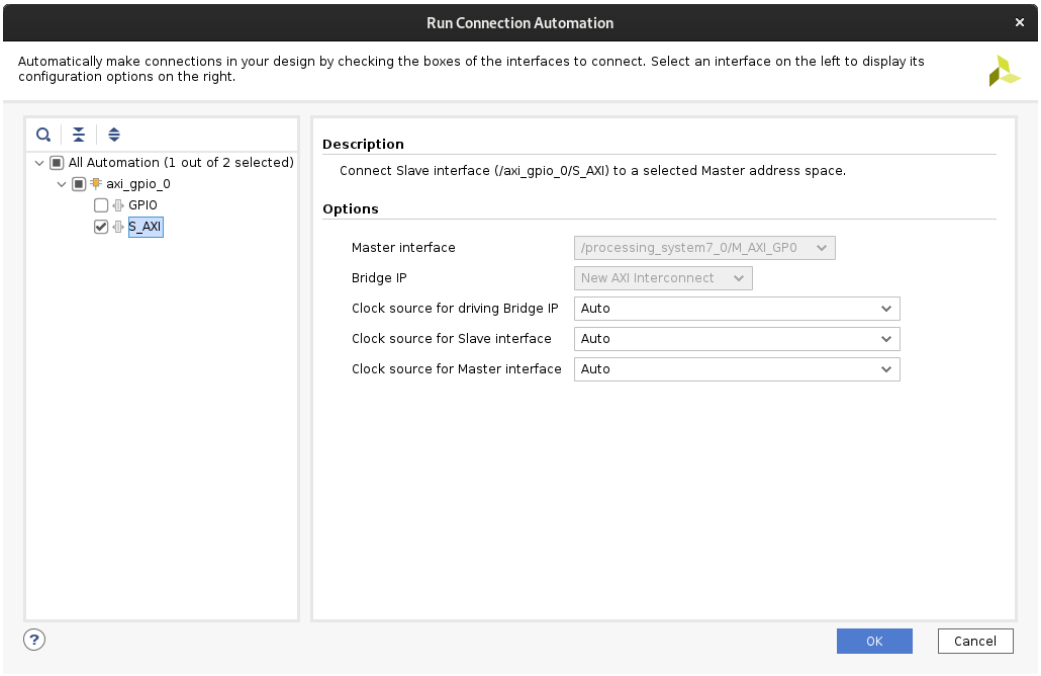
Bylo vytvořeno blokové schéma, které funguje jako výchozí modul a definuje propojení akcelerátoru s procesorem. Nejprve byl přidán blok Zynq7, který definuje rozhraní s procesorem, a byla spuštěna automatizace propojení komponent. Stav po této operaci je zobrazen na obrázku 4.1, na něm je vidět samotný blok a jeho základní propojení. Jedná se o propojení modulu k externí paměti a některým dalším pinům. Dále byl přidán blok GPIO AXI pro komunikaci po sběrnici. Před připojením na AXI je ale potřeba povolit AXI master sběrnici v nastavení bloku procesoru. Na obrázku 4.2 je zobrazeno příslušné nastavení, to se nachází v kategorii "AXI Non Secure Enablement"v záložce "PS-PL Configuration". Pak už lze GPIO na sběrnici připojit, nejjednodušší způsob je opět využít automatizace na propojení. Spuštění automatizace je zobrazeno na obrázku 4.3, zde se musí zvolit "S_AXI"automatizace. Automatizace "GPIO"na obrázku připojí GPIO blok na výstupní piny, proto nebylo spuštěno. Na obrázku 4.4 je výsledek běhu automatizace. GPIO blok je již připojený na AXI sběrnici a také byly vytvořeny další pomocné bloky, zejména propojovací síť, přes kterou se blok připojuje. Finální blockdesign je na obrázku 4.5. Do blokového schématu byl přidán blok "classifierInterface", který obsahuje VHDL implementaci zbytku akcelerátoru. Jeho rozhraní bylo navrženo se vstupem pro adresu (address), vstupu pro data (data_in), výstupu dat (data_out) a vstupu pro povolení zápisu (we). K bloku byl připojen blok GPIO tak, že jeho první kanál je připojen na vstup a výstup dat, používá tedy oba směry GPIO, jak je vidět na obrázku. Druhý kanál GPIO ovládá adresu a má jeden extra bit ovládající povolení zápisu. Pro rozdělení signálu na adresu a povolení zápisu se použily dva Slice bloky, které dovolují vybrat jakýkoli rozsah bitů ze signálu.



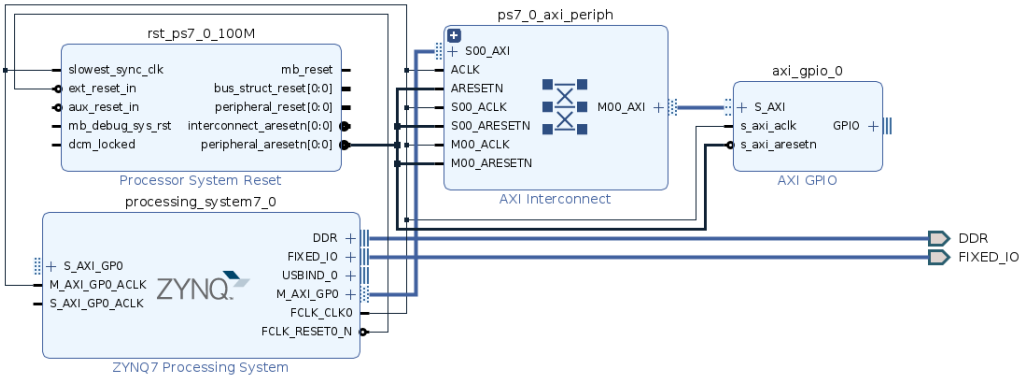
Obrázek 4.1: Blokové prostředí s blokem Zynq se základním propojením.



Obrázek 4.2: Povolení AXI v konfiguraci Zynq.

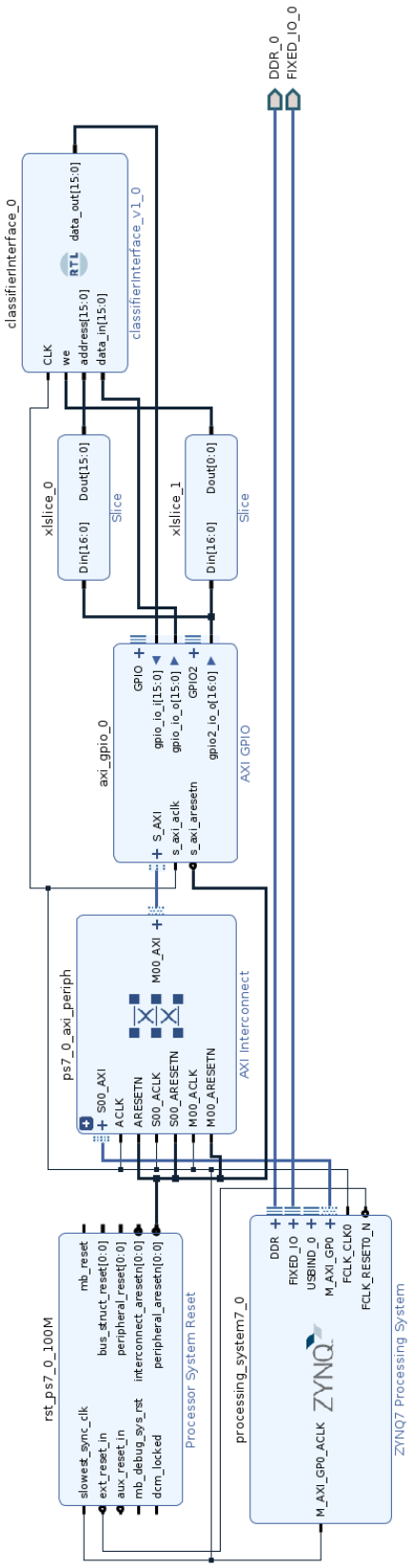


Obrázek 4.3: Spuštění automatického propojení.



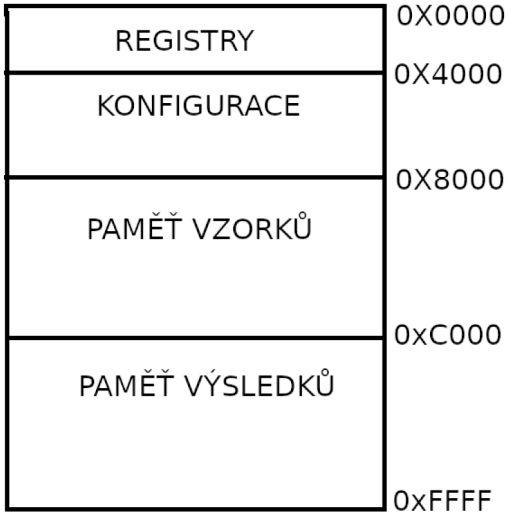
Obrázek 4.4: Výsledek automatického propojení.

Veškerá komunikace s akcelerátorem probíhá pomocí GPIO bloku, který je připojený na blok classifierInterface, jenž definuje komunikační rozhraní. Konkrétní zapojení bylo popsáno dříve v této kapitole. Toto rozhraní umožňuje procesoru číst data ze specifické adresy a to nastavením požadované adresy na vstup address a přečtením dat z data_out, kde budou platné další náběžnou hranu hodin dokud nedojde ke změně adresy. Rozhraní umožňuje také zápis dat na adresu zápisem dat na vstup data_in, nastavením adresy (address) a povolením zápisu vstupem we. K zápisu dojde při náběžné hraně hodin.



Obrázek 4.5: Pohled na konečný blockdesign.

Rozhraní classifierInterface poskytuje uniformní přístup k několika různým komponentám. Jsou jimi paměť vstupních dat, která se také označuje jako paměť vzorků, paměť výstupních dat, na níž se referuje také jako na paměť výsledků, a také řídicí signály a další konfigurovatelné komponenty. Všechny zmíněné komponenty jsou dostupné přes různé adresy. Pokud se jedná o paměť, její adresa je namapována do adresového prostoru rozhraní. Obsah adresového prostoru rozhraní je zobrazen na obrázku 4.6. Několik prvních adres je přiřazeno k registrům v rozhraní, registry umožňují zápis a čtení a jejich hodnoty jsou vyvedeny do řídicích signálů komponent dle potřeby konkrétní úlohy. Většina adres rezervovaných pro registry ale není využita, počet registrů lze jednoduše změnit podle potřeby. Adresy od 0x4000 do 0x8000 jsou adresy rezervované pro konfiguraci komponent, oproti registrům však rozhraní data neukládá a pouze předává data s adresou konfigurovatelné komponentě pro vlastní zpracování. V práci toto rozhraní bylo nakonec použito pro jednu konfigurovatelnou komponentu v podkapitole 4.3 pro programování pravidel klasifikace. Od adresy 0x8000 do 0xC000 je namapována paměť vzorků, do které procesor zapisuje vstupní data, a na zbytek adres je mapována paměť výsledků.



Obrázek 4.6: Paměťový prostor rozhraní s procesorem.

Rozdělení adresového prostoru je ve VHDL implementaci jednoduché, zejména proto, jakým způsobem byly navrženy počáteční adresy jednotlivých komponent. Pro výběr adresované komponenty byl vytvořen signál memory_select o šířce čtyři. VHDL proces nastavující tento signál je ve výpisu 4.1, který ukazuje jednoduché rozhodování podle dvou nejvyšších bitů adresy. Adresy se totiž liší pouze v patnáctém a šestnáctém bitu, což celý systém zjednodušuje. Každý bit signálu odpovídá jedné komponentě a je použitý pro výběr této komponenty. Adresovaná komponenta poté dostává adresu bez horních dvou bitů, tím pádem ji lze adresovat od nulové adresy.

```
memory_select_decode : process (address)
    begin
        case address(15 downto 14) is
            when "00" => memory_select <= "0001"; --registers
            when "01" => memory_select <= "0010"; --prog
            when "10" => memory_select <= "0100"; --samples
```

```
            when "11" => memory_select <= "1000"; --results
            when others => memory_select <= "0001";
        end case;
    end process;
```

Výpis 4.1: VHDL implementace procesu výběru adresované komponenty.

4.1.2 Paměť pro data

V systému jsou dvě paměti pro data, jedna pro zpracovávané vzorky a druhá pro výsledky zpracování. K oběma pamětem má přístup procesor a může do nich zapisovat a číst. Zároveň z paměti na vzorky se data čtou a předávají výpočetní komponentě akcelerátoru. Výsledky z výpočtu se pak vkládají do paměti výsledků. Paměti tedy musí mít dva přístupové porty, jeden pro procesor a jeden pro výpočetní modul. Všechny úlohy také vyžadují čtení více vzorků najednou pro rychlý výpočet. Jedna úloha vyžaduje zapisování více výsledků zároveň. To znamená, že paměti musejí mít nastavitelnou šířku, která určuje počet dat čtených a zapisovatelných najednou.

Na základě těchto požadavků byl vytvořen paměťový modul wide_ram.Modul wide_ram implementuje paměť se dvěma porty, jedním s šířkou nastavitelnou parametrem a druhým fixním. Modul je použitelný pro vzorky i pro výsledky. Protože paměť potřebuje uložit větší množství dat, používá k ukládání dat BlockRAM moduly. BlockRAM moduly mají dva nezávislé porty, které umožňují nastavit datovou šířku opět nezávisle. Úlohy však vyžadují poměrně velkou datovou šířku, například úloha z kapitoly 4.4 čte devět osmibitových vzorků najednou, což odpovídá sedmdesáti dvěma bitům, tak velkou datovou šířku BlockRAM neposkytuje. Bylo proto zapotřebí vytvořit paměť, která by takto velké šířky podporovala. Toho se docílilo použitím více BlockRAM modulů. Každý BlockRAM modul má datovou šířku jednoho vzorku a podle parametru specifikujícího požadovanou šířku ve vzorcích se vygeneruje počet BlockRAM. Adresa na širším portu modulu je přivedena na všechny BlockRAM zároveň a datové vstupy/výstupy jsou poskládány do jednoho vstupu/výstupu. U fixního portu, který používá procesor, je implementace složitější. Ve výpisu 4.2 je řešení čtení z fixního portu. V modulu se používají čtyři spodní bity adresy pro výběr jednotlivých paměťových modulů a zbytek adresy jako adresa do pamětí. Čtyři bity byly zvoleny, protože umožňují adresovat až šestnáct modulů, to znamená číst až šestnáct vzorků zároveň, což by mělo být dostatečné. Ve výpisu se používá smyčka for ke zpracování všech šestnácti výstupů, které jsou navázané na signál portAoutput. Tento signál má šířku šestnáct a pokud není potřeba maximální šířka, je vygenerováno menší množství BlockRAM a část signálu bude prázdná. To, jak je smyčka zapsána, při syntéze vytvoří multiplexor na výstupech pamětí řízený spodními bity adresy.

```
output_port_A : process (addrA, portAoutput)
    begin
        for i in 15 downto 0 loop
            if(addrA(3 downto 0) = i) then
                doA <= portAoutput(i);
            end if;
        end loop;
    end process;
```

Výpis 4.2: Řešení čtení z fixního portu.

Zápis do paměti se řeší podobným způsobem jako čtení. Ve výpisu 4.2 je implementace procesu čtení, u zápisu se používá stejný způsob dekódování adresy se smyčkou. Byl zaveden pomocný signál ram_select o šířce počtu BlockRAM, kde každý bit je zapojen na povolovací signál portu každé BlockRAM. Vstupní data jsou zapojena na každou BlockRAM, ale k zápisu dojde pouze u modulu, který byl vybrán.

4.2 Hledání řetězců v textu

První úloha se zabývá rychlým hledáním řetězců. Výsledná aplikace bude akcelerovat hledání jednoho ze specifikovaných řetězců v zadaném textu. Tento text bude předán programovatelné logice (PL) procesorem. Jako výsledek operace je pozice v textu, kde byl nalezen jeden z hledaných řetězců, případně informace, že žádný řetězec nalezen nebyl. V aplikaci půjde pomocí parametrů nastavovat požadované zrychlení vyhledávání a maximální počet hledaných řetězců.

Jak bylo řečeno v kapitole 3, úloha je navržena na základě jednoduchého algoritmu. Jedná se o jednoduchý, ale obecně pomalejší algoritmus oproti jiným algoritmům, které dokáží některé pozice přeskakovat. Tento algoritmus hledá řetězec na všech pozicích v textu. Začne na pozici prvního znaku v textu a ve smyčce pak porovnává každý znak z textu se znakem z hledaného řetězce, dokud se shodují. Pokud smyčka dojde na konec hledaného řetězce, řetězec byl na této pozici nalezen a výpočet končí, pokud se dříve naleznou neshodující se znaky, řetězec se na této pozici nenalézá, smyčka se přeruší a pokračuje se hledáním na další pozici. Tento algoritmus byl implementován v jazyku Python a je zobrazen ve výpisu 4.3. Tato implementace bude použita jako referenční řešení k ověření funkčnosti a k měření zrychlení.

```
def stringSearchAlgorithm(data, searchedString):
    idx = -1
    for i in range(0, len(data) - len(searchedString) + 1):
        for j in range(0, len(searchedString)):
            if(data[i + j] == searchedString[j]):
                if(j == len(searchedString) - 1):
                    idx = i
                    break
            else:
                break
        if(idx != -1):
            break
    return idx
```

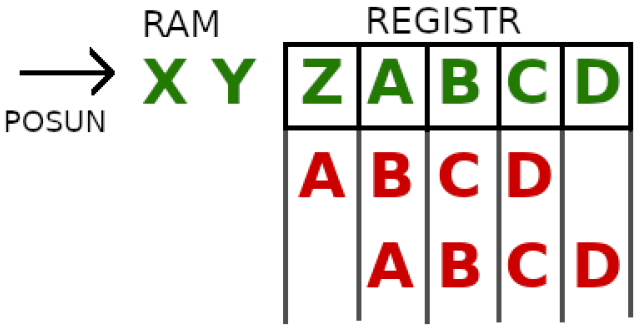
Výpis 4.3: Jednoduchý algoritmus hledání řetězce v Pythonu.

4.2.1 Architektura

Návrh akcelerátoru byl založen na dříve popsaném jednoduchém algoritmu. Princip fungování je takový, že se bude text vkládat do registru o délce hledaného řetězce. Text v registru se dále paralelně porovná s hledaným řetězcem. Dokud se řetězec nenalezne, v registru se pokaždé znaky posunou o pozici dál a na uvolněnou pozici se vloží další znak v pořadí.

V úloze se ještě používá kromě paralelního porovnávání další mechanizmus urychlení výpočtu. Princip fungování je naznačený na obrázku 4.7. Tento mechanizmus funguje tak,

že je více porovnávacích modulů, kde každý modul hledá stejný řetězec, ale na jiné pozici v registru. Na obrázku je registr a jeho obsah je naznačený jako tabulka s jedním řádkem a každý sloupec obsahuje jeden znak. Na obrázku jsou vyznačené dva porovnávací moduly jako dva červené řetězce (ABCD) pod registrem, na obrázku jsou také naznačeny pozice, na kterých moduly vyhledávají. Všechny moduly pracují paralelně a do registru se vkládá více znaků najednou, stejný počet jako je počet porovnávacích modulů, hledaný řetězec pak vždy nalezne jeden z modulů. A protože se pokaždé vkládá více znaků najednou, výpočet se zrychlí. To znamená, že každý modul urychlí výpočet o 100%. Počet modulů je nastavitelný parametrem.



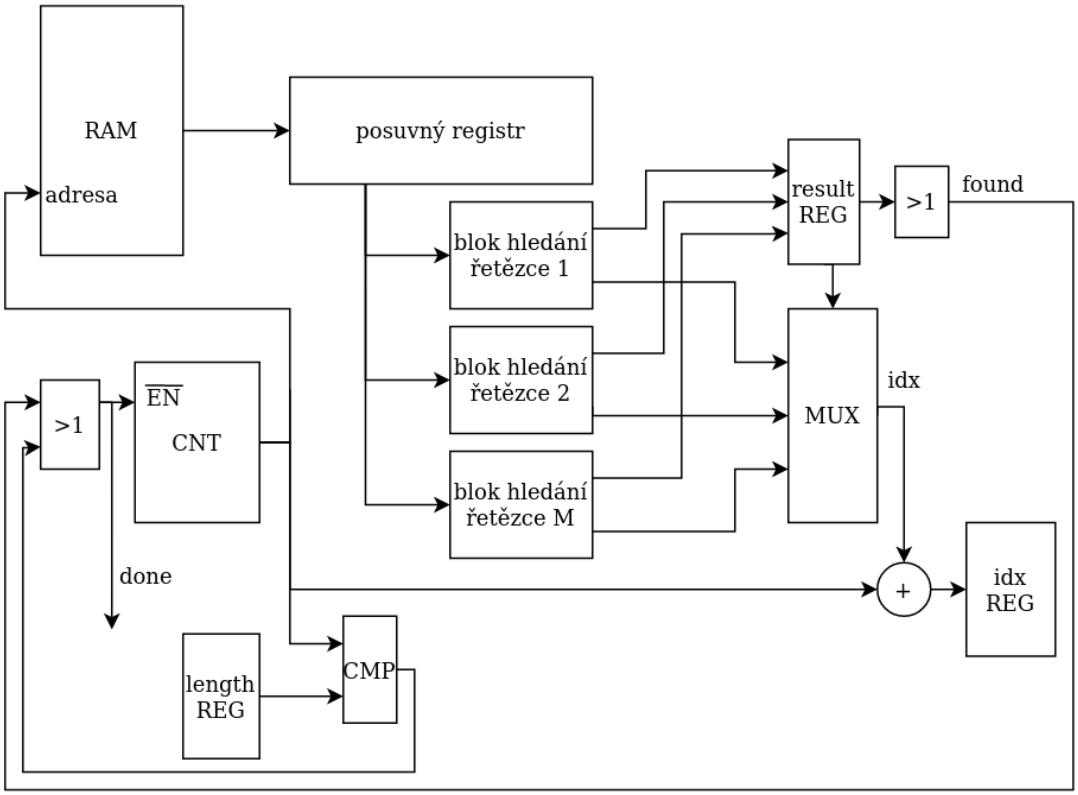
Obrázek 4.7: Princip urychlení výpočtu.

Celková architektura je zobrazena na obrázku 4.8. Princip spočívá ve čtení dat z paměti do posuvného registru. Registr by šlo označit také jako zřetězenou linku. Obsah registru se přivádí na vstup M bloků hledání řetězce, počet bloků je konfigurovatelný. Každý z těchto bloků vyhledává jeden řetězec v posuvném registru. Celkem lze tedy hledat až M řetězců najednou. Procesor může zjistit, který z řetězců byl nalezen, z registru "result REG", kde bude nastaven příslušný bit. Při nalezení hledaného řetězce se hledání zastavuje pomocí signálu "found" z OR hradla napojeného na všechny bity result registru. Signál found generuje signál "done". Ten zastaví čítač, který vkládá nová data z paměti RAM do posuvného registru. Po skončení výpočtu se do registru "idx REG" vloží pozice nalezeného řetězce v datech. Pozice se počítá z adresy, na které se čítač zastavil, a výstupu idx z bloku hledání řetězce, který udává začátek řetězce v posuvném registru. Pomocí multiplexoru se vybere hodnota idx z bloku, který řetězec našel. Pokud se žádný z hledaných řetězců v datech nevyskytuje, musí se hledání zastavit po průchodu všech dat. K tomu slouží "length REG". Do tohoto registru procesor specifikuje délku dat. Adresa z čítače se porovnává v komparátoru (CMP) z obsahem registru length a generuje signál done, který výpočet zastaví.

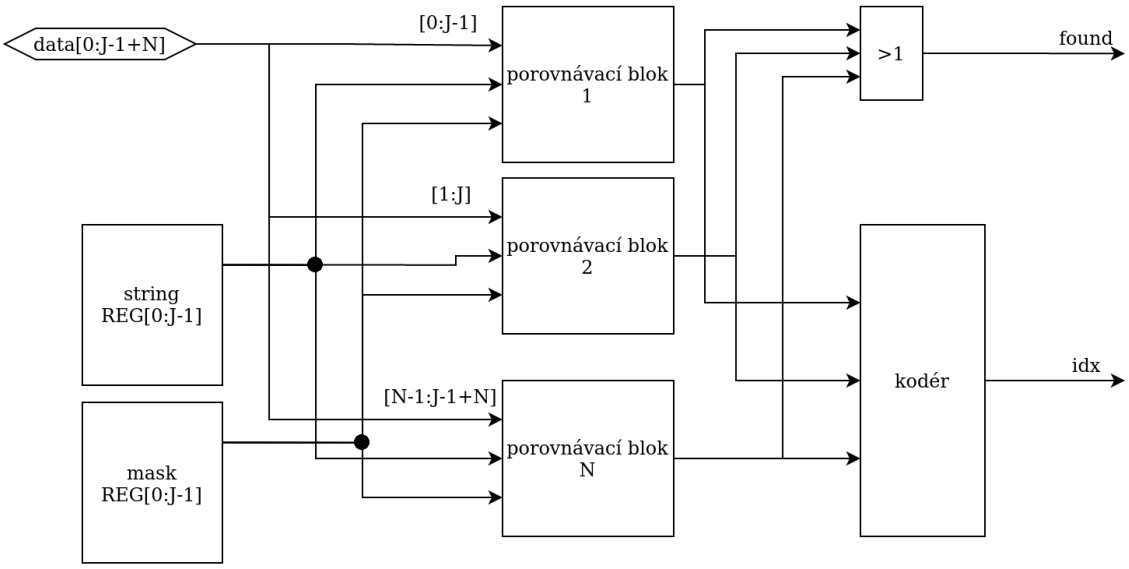
Blok hledání řetězce na obrázku 4.9 slouží k urychlení hledání řetězce paralelním hledáním. Pro hledání používá porovnávací bloky z obrázku 4.10. Používá N těchto bloků paralelně, přičemž hodnota N je konfigurovatelná. Čím více se použije bloků, tím rychlejší výpočet bude, avšak tím více se spotřebuje zdrojů. Hledaný řetězec je uložen v registru stringReg a maska řetězce v maskReg. Hodnoty obou registrů jsou přivedeny na vstup všech porovnávacích bloků. Vstupní data jsou část dat, v které se vyhledává. Jejich délka je rovna maximální délce hledaného řetězce plus počet porovnávacích bloků. Každý porovnávací blok je připojen na vstupní data posunuté o číslo indexu bloku znaků. První tedy není posunutý, druhý je posunutý o jeden a třetí je posunutý o dva znaky. Toto zapojení

vyhledává řetězec na N pozicích ve vstupních datech. Data na vstupu se tak mohou každý takt posunovat až o N znaků ve vyhledávaném textu. Dojde tak k N-násobnému zrychlení vyhledávání. Výstupní signály found z vyhledávacích bloků jsou agregované v OR hradlu na výsledný found signál bloku, který značí nalezení řetězce. K určení pozice hledaného řetězce to ale nestačí a musí se signalizovat, na které pozici byl řetězec nalezen, tedy jaký modul ho nalezl. Tato informace se kóduje do výstupu idx v kodéru, který pozici zakóduje do binárního kódu.

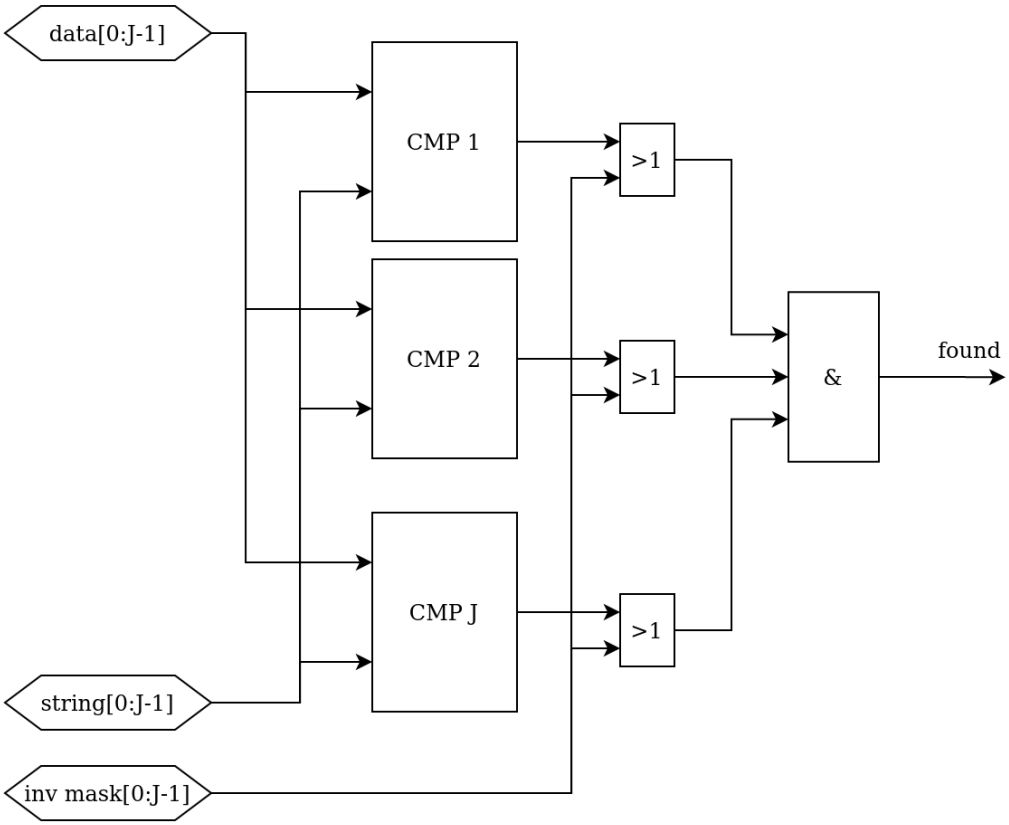
Porovnávací blok na obrázku 4.10 je základní komponenta pro hledání řetězců. Blok se skládá pouze z kombinační logiky. Má za úkol porovnat dva řetězce. Při syntéze obvodu bude známá maximální délka řetězce, ale skutečná délka bude specifikována až za běhu systému dynamicky pomocí masky. Modul má "J" komparátorů, což odpovídá maximální délce řetězce, kterou půjde před syntézou nakonfigurovat. Každý komparátor porovnává mezi sebou dva osmibitové vektory odpovídající jednomu znaku řetězců. Maska je do modulu přivedena invertovaná, tedy logická nula znamená platný znak a logická jednička neplatný. Jednotlivé bity masky se aplikují v logickém součtu s příslušnými výstupy komparátorů. To zajistí, že každý signál příslušející neplatnému znaku bude vždy v logické jedničce. Výsledek pak stačí redukovat v logickém součinu ve vícevstupém AND hradlu. Výstup je signál found značící, že vstupní řetězce se shodují. V celé úloze se používá pouze invertovaná maska. Inverze masky se provede už v procesoru a do registru masky bude uložena výsledná hodnota. Toto ušetří množství potřebných hradel, v opačném případě by se musela maska invertovat v hardwaru.



Obrázek 4.8: Základní struktura.



Obrázek 4.9: Blok hledání řetězce.



Obrázek 4.10: Porovnávací blok.

4.3 Klasifikace pomocí rozhodovacích stromů

Tato úloha se zabývá akcelerací klasifikace vzorků pomocí rozhodovacích stromů. Rozhodovací stromy se používají k rozdělení vzorků do tříd. Jelikož se jedná o demonstrační úlohu, byla zvolena jedna z jednodušších variant. Jako klasifikované vzorky se používají šestnáctibitová čísla a každý vzorek se klasifikuje do tříd, které jsou také šestnáctibitové.

Byla vytvořena referenční implementace v pythonu, která poslouží jako ilustrace algoritmu a také jako referenční řešení pro ověření funkčnosti akcelerátoru. Tato implementace je ve výpisu 4.4. Funkce přijímá v argumentech rozhodovací strom spolu se vzorky. Strom je uložen v jednorozměrném poli klasickým způsobem, tzn. levý potomek uzlu se nalézá na pozici 2*p+1 a pravý na 2*p+2, kde p je pozice rodičovského uzlu, a pozice kořene je na pozici nula. Funkce klasifikuje vzorky v poli samples a vrací pole o stejné velikosti s výsledky. Ve vnořené smyčce dochází k samotné klasifikaci vzorků. Každá iterace odpovídá jedné úrovni stromu, pomocí proměnné treePos, která obsahuje index aktuálního uzlu, se získá hodnota uzlu stromu a provede se porovnání. Podle výsledku porovnání dochází k přesunu do levého nebo pravého potomka. Po dosažení poslední vrstvy stromu obsahující třídy, nedochází k dalšímu porovnání, ale k zápisu hodnoty do výsledků.

```
def classify(tree, samples):
    results = []
    for sample in samples:
        treePos = 0
        for i in range(0, 3):
            if(sample <= tree[treePos]):
                treePos = (2 * treePos) + 1
            else:
                treePos = (2 * treePos) + 2
        results.append(tree[treePos])
    return results
```

Výpis 4.4: Referenční řešení klasifikace v pythonu.

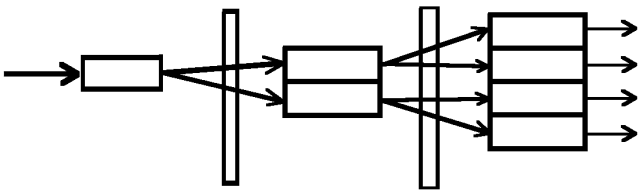
4.3.1 Architektura

Na obrázku 4.11 je zobrazen princip fungování klasifikace v akcelerátoru. Pro každou úroveň stromu je vyhrazena jedna paměť s klasifikačními pravidly. Na obrázku je zobrazen strom se třemi úrovněmi a v každé úrovni je paměť zobrazena jako soubor obdélníků symbolizujících řádky paměti. Vzorky putují stromem po jednotlivých úrovních, kde postupně probíhá klasifikace vzorku, a jelikož se jedná o binární strom, paměť se v každé úrovni zdvojnásobuje. V poslední úrovni stromu jsou místo pravidel uloženy v paměti třídy, které se pak ukládají do paměti s výsledky.

Samotná klasifikace probíhá tak, že vzorek se v každé úrovni porovná s hodnotou uloženou v paměti. Adresu do paměti pro porovnání vytváří předchozí úroveň. Každá úroveň počítá adresu podle výsledku porovnání a podle předchozí adresy tak, aby se zajistil stromový průchod pamětí. Na první úrovni, kde má paměť jenom jednu položku, se nastavuje počáteční adresa na nulu. Na dalších úrovních jsou výsledné adresy naznačeny na obrázku jako šipky.

Pro zvýšení propustnosti a také dodržení časování je každá úroveň stromu oddělena registry, ty jsou v obrázku značené jako podélné obdélníky mezi pamětmi. Mezi registry se

předávají klasifikované vzorky a adresy do pamětí jednotlivých úrovní. Jednotlivé úrovně stromu pak tvoří zřetězenou výpočetní linku. Další použitý způsob urychlení výpočtu spočívá ve vytvoření více stromů pracujících paralelně a v akcelerátoru je umožněno počet stromů nastavit parametrem. Potom zdvojnásobení počtu stromů znamená zdvojnásobení propustnosti akcelerátoru.



Obrázek 4.11: Princip klasifikace.

Třídy a pravidla pro klasifikaci akcelerátor umožňuje procesoru měnit za běhu. Procesor má také přístup do paměti akcelerátoru pro zápis vzorků a čtení výsledků klasifikace. Pro komunikaci akcelerátoru s procesorem byl použit komunikační systém z kapitoly 4.1. V kapitole byla popsána paměť wide_ram, která umožňuje nastavit parametrem šířku jednoho čtecího a zapisovacího portu. Paměť byla využita pro obě paměti, jak paměť pro vzorky, tak i pro paměť s výsledky. Šířka obou pamětí byla nastavena na šestnáct bitů krát parametr určující počet stromů, protože vstupní i výstupní data klasifikátoru jsou šestnáctibitové a každý strom potřebuje číst jeden vzorek a zapisovat jeden výsledek každý takt. Paměti wide_ram jsou už zpřístupněny procesoru pro čtení i zápis systémem pro komunikaci, jak je popsáno v kapitole 4.1.

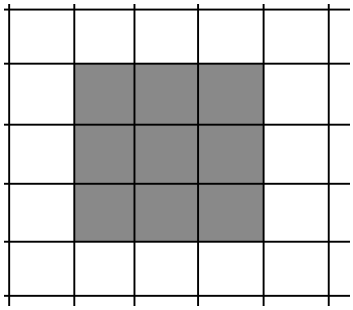
Klasifikační pravidla nejsou v akcelerátoru nastavena na pevno, ale je nutné je specifikovat ze strany procesoru. Na obrázku 4.11 je zobrazen princip fungování, který byl už dříve v této kapitole popsán. Klasifikace funguje na principu porovnávání vzorku s uloženými hodnotami, ty jsou označované jako klasifikační pravidla. Pravidla jsou uloženy v paměti akcelerátoru a pro jejich naprogramování bylo vytvořeno rozhraní. Rozhraní se skládá z adresy, na kterou jsou namapovány všechny paměti, vstupních dat a signálu povolujícího zápis. Rozhraní bylo zapojeno do systému popsaném v kapitole 4.1, aby k němu měl procesor přístup. V komunikačním systému bylo rozhraní zpřístupněno pro procesor na adresách od 0x4000 do 0x8000.

Systém pro komunikaci z kapitoly 4.1 poskytuje také několik registrů, do kterých má procesor přístup. Byly použity dva registry, jeden kam procesor specifikuje počet vzorků, který se má klasifikovat, a druhý pro řídicí signály spuštění výpočtu a resetování akcelerátoru.

Průběh výpočtu řídí jednoduchá logika, po nastavení signálu spuštění výpočtu se spustí adresový čítač, který adresuje paměť se vzorky. Vzorky se z paměti čtou a posílají na vstup rozhodovacím stromům. Vzorky se při klasifikaci pohybují ve zřetězené lince klasifikátoru, ta vytváří určité zpoždění výsledku. Pro určení adresy, kam výsledek zapsat do paměti výsledků, se musí adresa vzorku také zpozdit o stejnou dobu. Proto modul rozhodovacího stromu na vstupu přijímá kromě vzorku také adresu vzorku a na výstupu ji vrací jako adresu výsledku.

4.4 Mediánový filtr

Úloha se zabývá akcelerací mediánového filtru. Mediánový filtr se používá pro odstranění šumu ze signálu, zejména pro obrazová data. Princip filtru spočívá v tom, že každý bod v signálu je nahrazen mediánem hodnot z jeho okolí. Úloha řeší zpracování obrazových dat pomocí mediánového filtru. Při zpracování obrazu jsou body jednotlivé pixely. Obecně okolí může mít jakýkoli tvar, většinou však je čtvercové nebo kruhové. V navržené úloze se použilo čtvercové devíti-okolí. Na obrázku 4.12 je zobrazena část zpracovávaného snímku. Do devíti-okolí, se kterým úloha počítá, se počítají všechny přilehlé pixely včetně zpracovávaného pixelu, devíti-okolí jednoho pixelu je na obrázku zvýrazněné šedě. Pixely jsou složeny z jednoho osmibitového čísla, úloha tedy pracuje s černobílými snímky.



Obrázek 4.12: Část snímku s vyznačeným devíti-okolí.

U mediánového filtru se okolí bodu také říká okno. Bodům v signálu, kde nedojde k naplnění okna, protože signál není nekonečný, se říká okraj. Okraj je tedy definován velikostí a tvarem okna a v této úloze vychází okraj na hranu obrazu a má tloušťku jednoho pixelu. U filtru na tomto okraji vzniká problém, jak řešit okno zasahující mimo obraz. K problému se používá více přístupů. Mezi používané řešení patří například zmenšení okna na okrajích, vyplnění okna jinými body z přilehlé oblasti okna nebo nepočítání okrajů. V úloze se používá poslední zmíněné řešení, kde se okraje nepočítají a výsledný obraz je potom ořízнutý.

Byla vytvořena referenční implementace v pythonu, která poslouží jako ilustrace algoritmu a také jako referenční řešení k ověření funkčnosti akcelerátoru. Tato implementace je ve výpisu 4.5. Funkce přijímá filtrovanou fotku. Fotka je černobílá a každý pixel je jedno číslo. Fotka je předána funkci ve formě pole řádků, kde každý řádek je pole obsahující jednotlivé pixely. Funkce pak vrací fotku ve stejném formátu. Výpočet se skládá ze dvou smyček, které prochází fotku po pixelech, kromě okrajů. Do proměnné window se pak vloží celé devíti-okolí a pole se poté seřadí podle velikosti. Hodnota výsledného pixelu je dána prostřední hodnotou ze seřazeného seznamu, v devíti-okolí jde o pátý prvek.

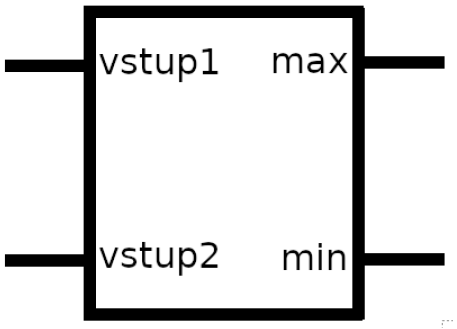
```
def medianFilter(image):
    resultImage = []
    rows = len(image)
    cols = len(image[0])
    for row in range(1, rows - 1):
        resultCol = []
        for col in range(1, cols - 1):
            window = []
            window.append(image[row - 1][col - 1])
```

```
            window.append(image[row - 1][col ])
            window.append(image[row - 1][col + 1])
            window.append(image[row ][col - 1])
            window.append(image[row ][col ])
            window.append(image[row ][col + 1])
            window.append(image[row + 1][col - 1])
            window.append(image[row + 1][col ])
            window.append(image[row + 1][col + 1])
            window.sort()
            resultCol.append(window[4])
        resultImage.append(resultCol)
    return resultImage
```

Výpis 4.5: Referenční řešení mediánového filtru v pythonu.

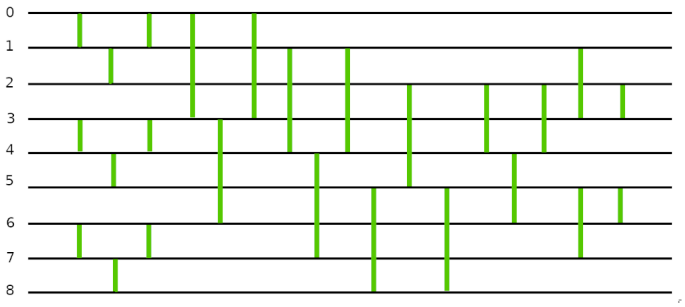
4.4.1 Architektura

Pro nalezení mediánu v okolí pixelu je potřeba pixely seřadit a vybrat prostřední hodnotu. Řazení se v úloze provádí pomocí řadicí sítě, která se skládá z porovnávacích bloků. Porovnávací blok je dvojvstupová komponenta s dvěma výstupy, schéma porovnávacího bloku je na obrázku 4.13. Vstupy bloku se mezi sebou porovnají a na výstup max se vloží větší a na výstup min menší z hodnot na vstupu. Vzájemným propojením bloků lze vytvořit řadicí síť o požadovaném počtu vstupů.

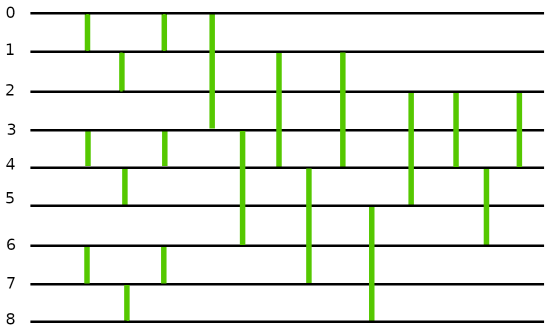


Obrázek 4.13: Porovnávací blok.

V úloze byla použita řadící sít s devíti vstupy, její zapojení je zobrazeno na obrázku 4.14. Na obrázku jsou vstupy znázorněné na levé straně a jsou očíslovány. Zelené vertikální čáry znázorňují porovnávací bloky. Vstupní data prostupují sítí zleva doprava a postupně dochází k řazení. Jelikož úkolem je pouze najít prostřední prvek (medián) oproti kompletnímu seřazení vstupů, byly ze sítě odstraněny nepotřebné porovnávací bloky. Některé bloky totiž nemají vliv na výsledný medián a lze je proto odstranit. Na obrázku 4.15 je výsledná použitá síť, která vznikla z řadící sítě na obrázku 4.14, ze které bylo odstraněno šest nepotřebných porovnávacích bloků.



Obrázek 4.14: Řadicí síť.



Obrázek 4.15: Použitá řadicí síť.

Pro urychlení výpočtu byl výpočet zřetězen oddělením některých bloků registry. Proto byla vytvořena zřetězená verze porovnávacích bloků, které mají navíc na vstupu hodinový signál a obsahují registry připojené na vstupech. Registry však musí být i na signálech, na které není v příslušném místě připojený žádný blok, aby se zajistilo stejné zpoždění na všech signálech. Pro dodržení časování na 100MHz byla potřeba vytvořit registry na každé druhé úrovni bloků.

V úloze je parametr určující úroveň paralelního zpracování. Parametr ovládá počet stupňů zřetězené linky. Více stupňů zřetězené linky znamená, že se zpracovává více vzorků zároveň, ale ke zrychlení nedojde. Rozdělením výpočtu na více stupňů se však docílí zvýšení maximální frekvence a zvýšením frekvence se zrychlení docílí. Jelikož v každém stupni zřetězené linky jsou dvě úrovně bloků, aby bylo dodrženo časování, jediná možnost byla dát do každého stupně jednu úroveň bloků a zdvojnásobit tak počet stupňů. Jiné rozdělení nedává smysl, jelikož ideální zrychlení se dá docílit pouze pokud jsou všechny stupně linky vyvážené. Proto má parametr pouze dvě možné hodnoty.

Protože úloha je pouze demonstrační a nejedná se o reálnou aplikaci mediánového filtru, jak bylo zdůvodněno v kapitole 3, jsou vstupní data předzpracovaná tak, že do paměti se nevkládají snímky, ale jednotlivá devíti-okolí snímků za sebou. Odpadá tak složitější procházení obrazu a stačí data číst z paměti po devíti hodnotách. Úloha se tak může soustředit čistě na proces hledání mediánu.

U akcelerátoru je potřeba, aby měl procesor přístup k pamětem akcelerátoru, konkrétně k paměti vzorků, kam bude zapisovat jednotlivé pixely, a k paměti výsledků, ze které bude výsledky číst. K tomuto účelu byl použit systém komunikace popsaný v kapitole 4.1. Z paměti na vzorky je potřeba číst celé devíti-okolí, proto byla použita paměť wide_ram (popsaná v 4.1), která byla nastavena na šířku devíti osmibitových vzorků. Filtr vytváří

pouze jeden výsledek v jednom taktu, proto stačilo na paměť s výsledky použít jenom BlockRAM modul nakonfigurovaný na datovou šířku vzorku.

Použitý systém pro komunikaci z kapitoly 4.1 poskytuje několik registrů, do kterých má procesor přístup. Stejně jako v úloze klasifikace (v kapitole 4.4) byly použity dva registry. Do jednoho registru procesor specifikuje počet vstupních dat a dva bity druhého registru jsou připojeny na řídicí signály spuštění výpočtu a resetování akcelerátoru.

Průběh výpočtu řídí jednoduchá logika, po nastavení signálu spuštění výpočtu se spustí adresový čítač, který adresuje paměť se vstupními daty. Jak bylo dříve v této kapitole zdůvodněno, do paměti se vkládají procesorem jednotlivé devíti-okolí za sebou. Vzorky se z paměti čtou a posílají na vstup výpočetnímu prvku. Vzorky se při výpočtu pohybují ve zřetězené lince, ta vytváří určité zpoždění výsledku. Pro určení adresy, kam výsledek zapsat do paměti výsledků, se musí adresa vzorku také zpozdit o stejnou dobu. Proto výpočetní modul na vstupu přijímá kromě vzorku také adresu vzorku a na výstupu ji vrací jako adresu výsledku.

Kapitola 5

Výsledky

V kapitole 4 byly navrženy tři úlohy, v této kapitole jsou prezentovány dosažené výsledky. Výkon vytvořených akcelerátorů byl naměřen a porovnán s referenčními softwarovými aplikacemi. Referenční aplikace byly měřeny na procesoru přípravku Pynq. Tím se demonstruje zrychlení, kterého se dá dosáhnout na přípravku při přesunutí řešení problému na FPGA.

Měření výkonu bylo provedeno měřením času výpočtu akcelerátorů a referenčních aplikací nad testovacími daty. Referenční aplikace byly měřeny na relativně velkém počtu dat, aby se vyloučily režije spojené se začátkem a koncem výpočtu. Akcelerátory byly měřené na menším množství dat, jelikož jsou limitované velikostí svých pamětí. Výsledky by však měly být dostatečně přesné i s menším počtem testovacích dat, protože výpočet na akcelerátoru je velmi předvídatelný oproti výpočtu na procesoru, kde může dojít k přepnutí kontextu, výpadku stránky paměti a dalším jevům.

Jako referenční aplikace, proti kterým se akcelerátory srovnávají, byly zvoleny aplikace napsané v jazyce C++ a Python. Implementace aplikací v C++ byla zvolena, aby byl efektivně využitý výkon procesoru. Aplikace v Pythonu byla zvolena, protože to je jazyk, kterým se Pynq platforma prezentuje.

V každé úloze byl vytvořen jeden akcelerátor a pro jejich měření byl vytvořen jednoduchý čítač, který byl do nich zakomponován. Čítač byl zpřístupněn pro čtení procesorem pomocí rozhraní popsaném v kapitole 4.1. Čítač měří počet taktů, které trvá výpočet, a jelikož frekvence FPGA je známá, lze spočítat čas trvání výpočtu. Všechny měření akcelerátorů byly provedeny na přípravku Pynq Z2 s frekvencí FPGA 100MHz. V práci se měří pouze doba samotného výpočtu, to znamená od první inkrementace adresy paměti se vstupními daty do posledního zapsaného výsledku. Jak bylo v kapitole 3 zdůvodněno, cílem je zrychlení výpočtu, proto se komunikace mezi procesorem a FPGA do času výpočtu nezapočítává.

Pro měření času výpočtu aplikací v C++ byl použit čítač s nejvyšší přesností pomocí třídy high_resolution_clock ze standardní knihovny. Způsob měření je zobrazen na výpisu 5.1. Na výpisu je zobrazena část programu měření času referenční aplikace hledání v textu. Ostatní aplikace v C++ používají stejný způsob měření. Měření probíhá zaznamenáním stavu čítače a voláním funkce, po navrácení z funkce se znovu zaznamená stav čítače, výsledný čas je rozdíl těchto hodnot, který se konvertuje do mikrosekund na posledním řádku výpisu. U všech aplikací byl měřen čas výpočtu pouze funkce s daným algoritmem, ostatní části aplikace jako zpracování argumentů a generování testovacích dat se provádí mimo funkci. Programy byly přeloženy pomocí gcc překladače na platformě Pynq a spouštěny v prostředí Pynq Linuxu, který je s Pynq dodávaný. A pro dosažení většího výkonu byly aplikace překládány s přepínačem O3.

```
auto start = std::chrono::high_resolution_clock::now();
int found =
    string_search_algorithm(buffer.data(), buffer.size(), "STRING");
auto end = std::chrono::high_resolution_clock::now();
auto int_s =
    std::chrono::duration_cast<std::chrono::microseconds>(end - start);
```

Výpis 5.1: Měření času v C++.

V kapitole 4 už byla popsána referenční řešení v jazyku Python, ale jednalo se pouze o samotné funkce provádějící výpočet. Popsané funkce byly použity pro vytvoření aplikací, které budou umožňovat funkce měřit. Aplikace byly při měření spouštěny s verzí Pythonu 3.6.5. V aplikacích je kromě zmíněných funkcí přidán generátor testovacích dat a kód pro měření času výpočtu. Generátory dat umožňují specifikovat parametrem aplikace počet náhodně vygenerovaných dat. Ve výpisu 5.2 je kód generátoru dat pro úlohu mediánového filtru, který vygeneruje jeden snímek. Ve výpisu jsou vidět smyčky iterující po řádcích a sloupcích, které generují náhodné pixely, a protože jsou pixely v akcelerátoru černobílé a osmibitové, jsou jejich hodnoty v rozsahu 0 až 255. Jako náhodný generátor byl použitý generátor ze základního Python modulu random. Velikost snímku se specifikuje v prvním argumentu programu. Generování dat pro ostatní úlohy je řešeno podobným způsobem jako ve výpisu 5.2 s tím rozdílem, že stačí vygenerovat pouze list náhodných čísel.

```
side = int(sys.argv[1])
image = []
for row in range(0, side):
    coldata = []
    for col in range(0, side):
        coldata.append(random.randrange(0, 255))
    image.append(coldata)
```

Výpis 5.2: Generování testovacích dat pro úlohu mediánového filtru.

Čas trvání výpočtu Python aplikací byl měřen pomocí funkce perf_counter z modulu time, který je vestavěným modulem v Pythonu. Kód, kterým byl měřen, je ve výpisu 5.3. Zmíněná funkce byla použita, protože používá nejpřesnější dostupný čítač. Ve výpisu je vidět uložení obou časových bodů, před a po volání měřené funkce a výsledný čas se vypočítá jako rozdíl časů a uloží se do proměnné timeDelta. V modulu existuje ještě funkce perf_counter_ns, která vrací čas v nanosekundách oproti sekundám a umožňuje tak vyhnout se nepřesnostem spojeným s typem float funkce perf_counter. perf_counter_ns je však dostupná pouze ve vyšší verzi Pythonu než je na Pynq předinstalovaná (funkce je dostupná od verze 3.7), bylo ale rozhodnuto, že instalování nové verze není zapotřebí, protože nepřesnosti se při větším počtu testovacích dat výrazně na výsledku neprojeví.

```
timePointStart = time.perf_counter()
foundAt = stringSearchAlgorithm(text, b"STRING")
timePointEnd = time.perf_counter()
timeDelta = timePointEnd - timePointStart
```

Výpis 5.3: Měření času výpočtu Python aplikací.

Výsledky měření byly zaznamenány do tabulek. V tabulce 5.1 jsou výsledky úlohy hledání řetězců, v tabulce 5.2 jsou výsledky pro úlohu klasifikace pomocí rozhodovacích stromů a v tabulce 5.3 jsou výsledky pro úlohu mediánového filtru. V každé tabulce jsou uvedeny

naměřené výsledky akcelerátoru v několika konfiguracích a porovnány s referenční aplikací v C++ a Pythonu. U každé implementace je uvedena velikost testovacích dat, nad kterými bylo provedeno měření, a naměřený čas výpočtu. Dále je pro jednodušší porovnání vypočítaný čas na vzorek v nanosekundách, ten byl spočten následovně: *čas výpočtu / velikost dat*. Také bylo uvedeno zrychlení oproti referenčnímu řešení v C++, to se počítá: *čas na vzorek C++ implementace / čas na vzorek počítané implementace*. Velikost dat je uvedena ve vzorcích, které mají v každé úloze jiný význam a jinou velikost. V úloze hledání řetězců jsou vzorky osmibitové znaky, v úloze klasifikace jsou vzorky klasifikovaná data a jsou šestnáctibitové a v úloze filtru jsou vzorky devíti-okolí a mají velikost devět bytů (72 bitů).

Testované verze akcelerátorů jsou v tabulkách 5.1 a 5.2 označeny počty modulů. Počet modulů u každé úlohy určuje míru paralelního zpracování, který se konfiguruje pomocí parametru. Jak fungují moduly v každé úloze, je popsáno v kapitole 4. Pro měření byly vybrány čtyři různé konfigurace modulů. Pro konkrétní počty modulů byly zvoleny mocniny dvojky. V tabulce 5.3 jsou výsledky úlohy mediánového filtru, jež jsou rozdělené podle binárního parametru, který určuje míru paralelního zpracování. Parametr určuje délku zřetězené linky, menší varianta má v jedné úrovni linky dvě úrovně porovnávacích bloků řadící sítě a větší varianta má v každé úrovni linky jednu úroveň bloků. V tabulce jsou výsledky obou dvou možných konfigurací.

Úloha hledání řetězců				
Implementace	Velikost dat (vzorky)	Čas výpočtu	Čas na vzorek (ns)	Zrychlení oproti C++
akcelerátor (1 modul)	21	370 ns	17.619	0.566
akcelerátor (2 moduly)	21	170 ns	8.095	1.232
akcelerátor (4 moduly)	21	70 ns	3,33	2,99
akcelerátor (8 modulů)	21	30 ns	1.429	6.9811
C++	30 M	299,196 ms	9,973	1
Python	1 M	7,08811 s	7088,11	0,0014

Tabulka 5.1: Porovnání výsledků úlohy hledání řetězců.

Z měření je zřejmé, že akcelerátory jsou výrazně rychlejší než softwarové aplikace. Akcelerátory dosáhly zrychlení v řádech jednotek až desítek násobků oproti implementacím v C++, zatímco zrychlení oproti implementacím v Pythonu jsou řádově tisícinásobné. Nejmenší zrychlení oproti referenci v C++ dosáhl akcelerátor úlohy hledání v textu, kdy v konfiguraci s jedním modulem byl pomalejší než referenční aplikace, přesto ale akcelerátor dosahuje podstatného zrychlení v konfiguracích se čtyřmi a osmi moduly. Procesor je rychlejší, protože softwarovému řešení algoritmu hledání stačí velmi jednoduchý výpočet (jedno porovnání) na zpracování jednoho znaku oproti složitějším úlohám, procesor tak může využít svojí výhodu vyšší frekvence hodin oproti akcelerátoru. Aby akcelerátor dosáhl lepšího výsledku, musí lépe využít paralelního zpracování dat než konfigurace s jedním modulem.

Pro implementované akcelerátory byly změřeny použité hardwarové zdroje. V tabulce 5.4 jsou uvedeny využité zdroje pro akcelerátor hledání řetězců. V tabulce 5.5 jsou uvedeny

Úloha klasifikace pomocí rozhodovacích stromů				
Implementace	Velikost dat (vzorky)	Čas výpočtu	Čas na vzorek (ns)	Zrychlení oproti C++
akcelerátor (1 modul)	104	1080 ns	10,385	6,049
akcelerátor (2 moduly)	104	560 ns	5,384	11,6663
akcelerátor (4 moduly)	104	300 ns	2,885	21,772
akcelerátor (8 modulů)	104	170 ns	1,635	38,417
C++	30 M	1884,345 ms	62,8115	1
Python	1 M	9,94396 s	9943,96	0,006316548

Tabulka 5.2: Porovnání výsledků úlohy klasifikace pomocí rozhodovacích stromů.

Úloha mediánový filtr				
Implementace	Velikost dat (vzorky)	Čas výpočtu	Čas na vzorek (ns)	Zrychlení oproti C++
akcelerátor (menší zřetězení)	7	140 ns	20	26,54411
akcelerátor (větší zřetězení)	7	210 ns	30	17.69607
C++	10 M	5308,822 ms	530,8822	1
Python	1 M	25,96179 s	25961,7915	0,020448597

Tabulka 5.3: Porovnání výsledků úlohy mediánového filtru.

využité zdroje akcelerátoru klasifikace a v tabulce 5.6 jsou uvedeny využité zdroje akcelerátoru mediánového filtru. Ve sloupcích jsou uvedeny jednotlivé konfigurace stejného akcelerátoru a jejich spotřebované zdroje. Význam různých konfigurací akcelerátorů byl popsán dříve v kapitole a přesněji je popsán v kapitole 4. Všechny akcelerátory byly syntetizované pomocí aplikace Vivado verze v2021.1 (64-bit) SW Build: 3247384.

Využití zdrojů				
počet modulů	LUT	LUTRAM	FF	BRAM
1	1205	60	1880	0,5
2	1277	60	1837	0,5
4	1359	60	1858	1
8	1529	60	1827	2

Tabulka 5.4: využití zdrojů akcelerátoru hledání řetězců.

U všech akcelerátorů byl pozorován nárůst zdrojů se zvyšováním paralelního zpracování pomocí konfigurace parametru, kromě akcelerátoru filtru v tabulce 5.6, kde došlo k malému

Využití zdrojů				
počet modulů	LUT	LUTRAM	FF	BRAM
1	694	63	1090	2
2	769	75	1142	4
4	921	99	1246	8
8	1281	147	1458	16

Tabulka 5.5: Využití zdrojů akcelerátoru klasifikace.

Využití zdrojů				
zřetězení	LUT	LUTRAM	FF	BRAM
menší	857	69	1123	10
větší	849	85	1315	10

Tabulka 5.6: Využití zdrojů akcelerátoru mediánového filtru.

snížení využitých LUT. Zvýšení paralelního zpracování v tomto akcelerátoru se dosahuje rozdělením výpočtu na více úrovní přidáním registrů, což vysvětluje, proč se využití LUT nezvýšilo, lehké snížení by mohlo být vysvětleno tím, že změna umožnila jiné efektivnější mapování funkcí na LUT.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat tři úlohy prezentující hardwarovou akceleraci na přípravku Pynq Z2. Akcelerátory byly navrženy tak, aby bylo možné pomocí parametru ovládat míru paralelního zpracování a množství spotřebovaných zdrojů. Úlohy jsou vytvořeny pro výukové účely tak, aby byly dobře pochopitelné pro studenty. Práce probíhala podle zadání a všechny body se podařilo splnit.

Po prostudování přípravku Pynq Z2 a seznámení se s vývojovými nástroji pro technologii Xilinx Zynq, které jsou popsané v kapitole 2, jsem začal pracovat na vytvoření sady tří úloh. V každé úloze byl vytvořen hardwarový akcelerátor a obslužná aplikace na procesor. Akcelerátor byl implementován v jazyku VHDL a obslužná aplikace v jazyku Python.

Návrh vytvořených akcelerátorů je popsán v kapitole 4. První úloha akceleruje vyhledávání v textu. Jejími vstupy jsou vyhledávané slovo a samotný text, ve kterém se slovo hledá. Výsledkem vyhledávání je pozice nalezeného slova v textu, nebo informace, že se slovo v textu nenalézá. Druhá úloha řeší klasifikaci dat, jedná se o rozdělení dat do tříd podle zadaných pravidel. Vstupem úlohy jsou třídicí pravidla a samotná data. Výsledkem klasifikace je třída, do které vstupní data patří. Poslední úlohou je obrazový filtr. Jde o mediánový filtr, který se používá k odstranění šumu ve fotkách a videích.

Výsledné akcelerátory byly úspěšně ověřeny v simulacích a na přípravku Pynq Z2 a dosažené výsledky jsou popsané v kapitole 5. Kromě ověření funkčnosti byla u akcelerátorů naměřena rychlost výpočtu a porovnána s referenčními softwarovými aplikacemi. Všechny akcelerátory byly měřeny s různou konfigurací parametru, který určuje míru paralelního zpracování. Oproti softwarové implementaci napsané v jazyku C++ dosahovaly akcelerátory zrychlení v řádu jednotek až desítek násobků. V kapitole 5 jsou také uvedeny spotřebované zdroje FPGA pro každou měřenou konfiguraci akcelerátoru. Na závěr své práce jsem vytvořil krátký návod, aby si studenti mohli vytvořené implementace vyzkoušet.

Práce by mohla být v budoucnu rozšířena o další úlohy, například o úlohy zaměřené na efektivní přenos dat mezi procesorem a akcelerátorem. Ve vypracovaných úlohách se přenos dat řeší, ale není to hlavní zaměření úloh, a proto bylo zvoleno pomalejší, ale pro studenty názornější řešení.

Literatura

[1] Akbari, A.: Drivers of economic and financial integration: A machine learning approach. 2022, [Online; navštíveno 13. 5. 2022].
URL https://www.researchgate.net/figure/Schematic-of-a-Decision-Tree-The-figure-shows-an-example-of-a-decision-tree-with-3_fig1_348456545

[2] Arm: An introduction to AMBA AXI. 2022, [Online; navštíveno 10. 5. 2022].
URL https://developer.arm.com/documentation/102202/0200/Channel-transfers-and-transactions

[3] Rene Mueller, G. A., Jens Teubner: Sorting networks on FPGAs. *The VLDB Journal*, 2012: str. 1–23.
URL https://doi.org/10.1007/s00778-011-0232-z

[4] Xilinx: Zynq-7000 SoC. 2021, [Online; navštíveno 20. 11. 2021].
URL https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable

[5] Xilinx: 7 Series DSP48E1 Slice. 2022, [Online; navštíveno 4. 1. 2022].
URL https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[6] Xilinx: 7 Series FPGAs Configurable Logic Block. 2022, [Online; navštíveno 4. 1. 2022].
URL https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[7] Xilinx: 7 Series FPGAs Memory Resources. 2022, [Online; navštíveno 4. 1. 2022].
URL https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[8] Xilinx: AXI Basics 1 - Introduction to AXI. 2022, [Online; navštíveno 10. 5. 2022].
URL https://support.xilinx.com/s/article/1053914?language=en_US

[9] Xilinx: PYNQ- Python Productivity for Zynq. 2022, [Online; navštíveno 13. 1. 2022].
URL https://www.xilinx.com/support/university/boards-portfolio/xup-boards/XUPPYNQ-Z2.html