Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Vid Smole

# Izdelava pospeševalnika za izračun MD5 na napravi FPGA s knjižnico PYNQ

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Nejc Ilc
Somentor: doc. dr. Ratko Pilipović

Ljubljana, 2023



*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

**Kandidat:** Vid Smole

**Naslov:** Izdelava pospeševalnika za izračun MD5 na napravi FPGA s knjižnico PYNQ

**Vrsta naloge:** Diplomska naloga na univerzitetnem programu prve stopnje Računalništvo in informatika

**Mentor:** doc. dr. Nejc Ilc

**Somentor:** doc. dr. Ratko Pilipović

**Opis:**

Odprtokodni projekt PYNQ podjetja AMD predstavlja enostaven vmesnik do izrabe visoko zmogljivih programirljivih čipov FPGA preko programskega jezika Python. V diplomskem delu predstavite knjižnico PYNQ na primeru implementacije vezja za časovno učinkovit izračun zgoščenih vrednosti MD5. Na osnovi svoje izkušnje pri delu s knjižnico PYNQ tudi podajte oceno njene primernosti oziroma zrelosti za uporabo v širši skupnosti razvijalcev, ki nimajo poglobljenega znanja o napravah FPGA.

**Title:** MD5 accelerator on FPGA with the PYNQ library

**Description:**

The open-source project PYNQ by AMD provides a simple interface for harnessing high-performance programmable FPGA chips using the Python programming language. In the thesis, present the strong and weak points of the PYNQ library through the implementation of a circuit for time-efficient execution of the MD5 hashing algorithm. Based on your experience working with the PYNQ library, you should also assess its suitability or maturity for use in the broader community of developers who may not have in-depth knowledge of FPGA devices.

*Zahvaljujem se mentorjema doc. dr. Nejcu Ilcu in doc. dr. Ratku Pilipoviću za usmerjanje tekom pisanja ter za vse vsebinske in slovnične popravke. Zahvaljujem se tudi staršem za potrpljenje in podporo tekom študija ter sošolcem, ki so mi vedno bili pripravljeni pomagati.*

# Kazalo

# Seznam uporabljenih kratic

| kratica | angleško | slovensko |
|---|---|---|
| **ASIC** | application specific integrated circuit | aplikacijsko specifično integrirano vezje |
| **FPGA** | field-programmable gate array | programirljivo polje logičnih vrat |
| **IP** | intellectual property | intelektualna lastnina |
| **SoC** | system on chip | sistem na čipu |
| **GPIO** | general-purpose input/output | splošno-namenski vhod/izhod |
| **LUT** | lookup table | vpogledna tabela |
| **VLNV** | vendor, library, name, and version | proizvajalec, knjižnica, ime in različica |
| **PS** | programmable logic | programirljiva logika |
| **PL** | processing system | procesni sistem |
| **DMA** | direct memory access | neposredni dostop do pomnilnika |
| **FIFO** | first in, first out | prvi noter in prvi ven |
| **HDL** | hardware description language | jezik za opis strojne opreme |
| **HLS** | high-level synthesis | visokonivojska sinteza |
| **VHDL** | very high speed integrated circuit HDL | jezik za opis zelo hitrih digitalnih vezij |
| **ROM** | read-only memory | bralni pomnilnik |

# Povzetek

**Naslov:** Izdelava pospeševalnika za izračun MD5 na napravi FPGA s knjižnico PYNQ

**Avtor:** Vid Smole

V okviru diplomske naloge smo se želeli bolje spoznati z odprtokodno knjižnico PYNQ in podati svojo oceno njene primernosti za splošno uporabo v aplikacijah visoko zmogljivega računanja. Knjižnica nam omogoča pisanje vmesnikov za komunikacijo z logiko na čipu FPGA iz udobja jezika z veliko stopnjo abstrakcije Python. Kot primer uporabe knjižnice smo izbrali implementacijo cevovodnega pospeševalnika za izračun zgoščenih vrednosti z algoritmom MD5. Pospeševalnik sprejema in pošilja podatke s protokolom AXI in uporabo DMA, za zgoščevanje podatkov pa uporablja 64-stopenjski cevovod. Pokazali smo, da je izračun zgoščenih vrednosti z uporabo našega pospeševalnika hitrejši kot z uporabo standardne knjižnice, pri tem pa ne žrtvujemo kakovosti uporabniške izkušnje.

**Ključne besede:** FPGA, PYNQ, AXI, MD5, načrtovanje digitalnih sistemov, VHDL.

# Abstract

**Title:** MD5 accelerator on FPGA with the PYNQ library

**Author:** Vid Smole

The aim of this thesis was to familiarize ourselvers with the open source library PYNQ and to give our assessment of its suitability for general use in high-performance computing applications. The library allows us to write interfaces to communicate with logic on an FPGA from the comfort of the high-level abstraction language Python. As an example of the use of the library, we have chosen to implement a pipelined accelerator for the hashing algorithm MD5. The accelerator receives and sends data using the AXI protocol and DMA. It uses a 64-stage pipeline for data hashing. We have shown that computing hash values using our accelerator is faster than using the standard library, without sacrificing user experience.

**Keywords:** FPGA, PYNQ, AXI, MD5, digital systems design, VHDL.

# Poglavje 1

# Uvod

V zadnjih letih smo priča velikemu napredku na področju velikih jezikovnih modelov, računalniškega vida, strojnega prevajanja in mnogih drugih področjih, ki se ukvarjajo z obdelavo velikih količin podatkov [8]. Gonilo tega napredka so grafične procesne enote (GPE), saj te omogočajo izjemno hitro vzporedno obdelavo podatkov.

Hkrati živimo v obdobju, ki ga zaznamuje vse večja zaskrbljenost glede porabe energije in vpliva na okolje. Pri tem se na mnogih področjih kot alternativa energijsko potratnim grafičnim karticam ponujajo aplikacijsko specifična integrirana vezja (angl. *application specific integrated circuit*, ASIC). Ta se delijo na več vrst, od katerih so za nas najbolj zanimiva programirljiva logična vezja (angl. *field-programmable gate array*, FPGA), saj jih lahko hitro in poceni reprogramiramo in tako prilagajamo specifičnim nalogam [9, 3].

Ena največjih omejitev pri bolj splošni in razširjeni uporabi tehnologije FPGA je bilo do sedanje pomanjkanje orodij ter knjižnic, ki bi končnemu uporabniku (na primer podatkovnemu znanstveniku) omogočali preprosto uporabo pospeševalnika na čipu FPGA. Pri uporabi GPE, CPE in podobnih arhitektur lahko uporabnik problem v celoti reši z uporabo jezika z veliko stopnjo abstrakcije, kot je na primer Python, saj je zanj napisanih veliko knjižnic, ki zakrijejo kompleksnost implementacije [11]. Podjetje Xilinx, ki ga je

pred kratkim kupil AMD, želi podobno uporabniško izkušnjo ponuditi tudi za vezja FPGA, zato so napisali knjižnico PYNQ, ki omogoča uporabo funkcij, implementiranih na vezju FPGA, iz udobja programskega jezika Python v okolju Jupyter. Tak program lahko teče tako na cenovno relativno ugodni razvojni plošči, ki združuje vezje FPGA in mikroprocesor, kot tudi pospeševalnikih iz družine Alveo, ki jih med drugim lahko uporabljamo tudi preko ponudnika oblačnih storitev Amazon Web Services [4].

Namen te diplomske naloge je ugotoviti ali je knjižnica PYNQ tista, ki bo uporabo tehnologije FPGA prinesla tudi v domeno podatkovne znanosti, podobno kot se dandanes uporabljajo grafične procesne enote. Knjižnico smo uporabili za izdelavo vmesnika, preko katerega končni uporabnik uporablja pospeševalnik za izračun zgoščenih vrednosti z algoritmom MD5 na vezju FPGA. Vezje smo s protokolom AXI povezali z mikroprocesorjem na razvojni plošči in z knjižnico PYNQ omogočili uporabo v okolju Jupyter. Zrelost knjižnice in njeno primernost za bolj splošno uporabo smo ovrednotili subjektivno. Zanimalo nas je predvsem, kakšna je stopnja zahtevnosti dela s knjižnico za nas kot razvijalce pospeševalnika in če bomo lahko z uporabo te knjižnice končnemu uporabniku omogočili preprosto uporabo našega pospeševalnika, primerljivo z uporabo standardne knjižnice.

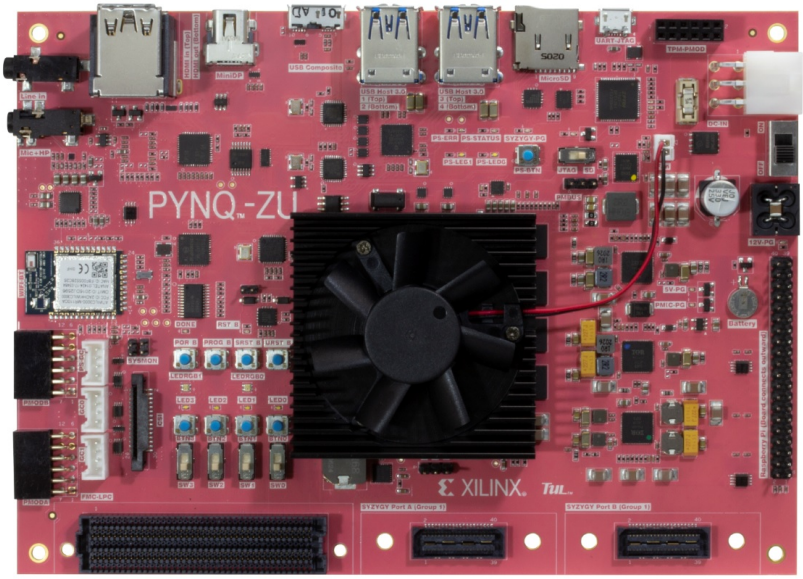
# Poglavje 2

# Projekt PYNQ

PYNQ je projekt, s katerim želi podjetje Xilinx širši množici uporabnikov približati tehnologijo FPGA [16]. Ugotovili so, da se na področju podatkovne znanosti zelo pogosto uporabljajo programski jeziki z veliko stopnjo abstrakcije strojne opreme, kot je na primer Python, ki s pomočjo raznih knjižnic skrijejo kompleksnost implementacije algoritma in tako poenostavijo in pospešijo razvoj programskih rešitev. Ti algoritmi so ponavadi napisani v jezikih z manjšo stopnjo abstrakcije strojne opreme, kot sta na primer C in C++ in pogosto tečejo na večjedrnih procesorjih ali grafičnih karticah, saj te omogočajo hitro in učinkovito obdelavo velikih količin podatkov. Tako uporabnik v jeziku z veliko abstrakcije strojne opreme poljubno uporablja funkcije, ki jih ponuja knjižnica, tudi če se ne spozna na programiranje porazdeljenih sistemov, grafičnih kartic in podobnih tehnologij.

Projekt PYNQ nam z istoimensko knjižnico omogoča, da na enak način uporabljamo tudi logiko, ki je implementirana znotraj vezja FPGA. Obnaša se kot most med programsko opremo (npr. okolje Jupyter) in strojno opremo (digitalno vezje na FPGA). V ozadju samodejno naloži implementirano vezje na čip FPGA, pripravi gonilnike, prepozna jedra IP (angl. *intellectual property*), ki bodo izvajala željeni algoritem in podobno. To omogoča uporabniku, da logiko na čipu FPGA uporabi s preprostim klicem funkcije, enako kot pri uporabi standardne knjižnice.

Da bi tehnologijo FPGA kar se da približali uporabnikom, so po zgledu projekta Raspberry Pi naredili razvojne plošče, ki se prav tako imenujejo PYNQ. Te temeljijo na Xilinxovi družini vezij SoC (angl. *system on chip*), imenovanih Zynq, ki na enem čipu združujejo procesor ARM in čip FPGA. Plošče PYNQ procesorju in čipu FPGA dodajo še kopico različnih vmesnikov, kot so na primer HDMI, USB, Ethernet, UART, splošnonamenski vhodi in izhodi (angl. *general-purpose input/output*, GPIO) in podobno, s čimer želijo nagovoriti tudi tiste, ki se ukvarjajo predvsem s programiranjem vgrajenih sistemov.




Slika 2.1: Razvojna plošča PYNQ-ZU[1].

V tej diplomski nalogi bomo uporabljali razvojno ploščo PYNQ-ZU, ki je prikazana na sliki 2.1. Ta plošča ima sistem na čipu Zynq Ultrascale+ XCZU5EG, ki vsebuje štirijedrni procesor ARM Cortex A53, FPGA s 117 000 vpoglednimi tabelami (angl. *lookup table*, LUT) in 4 GB DDR4 RAM[2]. Poleg

[1]Vir: `https://www.tulembedded.com/FPGA/ProductsPYNQ-ZU.html` (pridobljeno 6. 6. 2023).

[2]Bolj obsežne specifikacije so na voljo na `https://xilinx.github.io/PYNQ-ZU/overview.html`.

tega sistem na čipu vsebuje še dodaten procesor za realnočasovne aplikacije, sama plošča pa, kot je razvidno iz slike, veliko različnih vhodov in izhodov, ki jih v tej diplomski nalogi ne bomo uporabili.

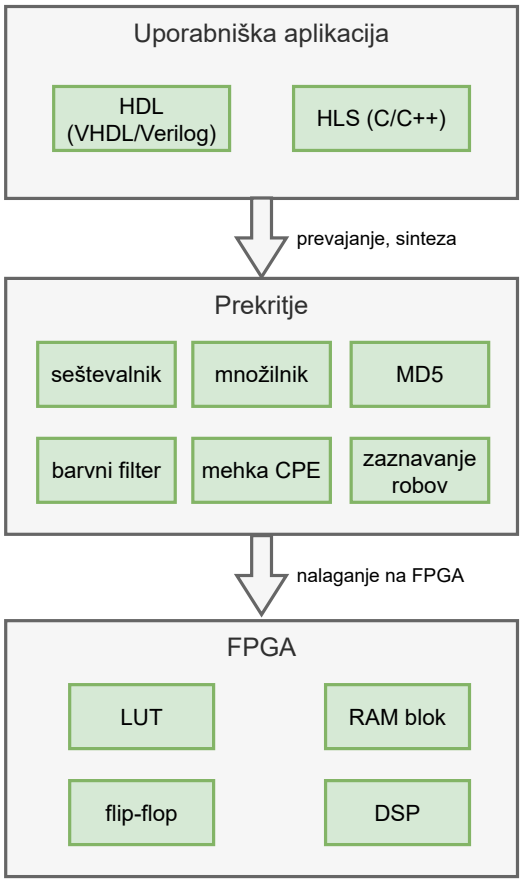
Knjižnico PYNQ lahko uporabljamo tudi brez razvojne plošče PYNQ, in sicer z uporabo pospeševalnikov iz družine Xilinx Alveo ali s poljubno drugo razvojno ploščo, ki ima SoC Zynq. Za mnoge je celotno okolje PYNQ že pripravljeno vnaprej, za ostale pa dokumentacija[3] opisuje postopek priprave kartice SD.

## 2.1 Prekritje

Ključni koncept s katerim deluje PYNQ je prekritje (angl. *overlay*). Arhitektura prekritja, ki je prikazana na sliki 2.2, doda nivo abstrakcije nad nivo digitalnega vezja na čipu FPGA. Na tem novem nivoju so večje in bolj specializirane komponente, ki so delne rešitve pogostih problemov, kot neke vrste strojne knjižnice. Te komponente so napisane kot jedra IP v jeziku HDL (angl. *hardware description language*) ali izdelana po postopku HLS (angl. *high-level synthesis*) in so implementirane s tipičnimi gradniki vezij na čipih FPGA (LUT, DRAM, flip-flopi itd.). Uporabnik lahko jedra IP poveže v celoto, ki reši nek problem ali pa jih združi v večja, kompleksnejša jedra in nato ta uporabi za reševanje problema. Primeri delnih rešitev, ki bi jih lahko implementirali kot jedra IP so na primer seštevalnik, množilnik ali deli cevovoda za obdelavo slik (barvni filter, zaznavanje robov, stiskanje slik itd). Poleg preprostih komponent lahko implementiramo tudi mehke CPE (angl. *soft processor*) ali celo mehke GPE [7]. V tej diplomski nalogi bomo implementirali jedro IP za izračun zgoščenih vrednosti z algoritmom MD5.

Prednost take arhitekture je hitrejši in enostavnejši razvoj, saj je povezovanje jeder IP enostavnejše kot načrtovanje celotne logike v jeziku HDL ali po postopku HLS, hkrati pa nam ni potrebno čakati več deset minut na sintezo in implementacijo, saj so jedra IP že pripravljena na uporabo.

[3]Dokumentacija je dostopna na naslovu `http://www.pynq.io/board.html`.




Slika 2.2: Arhitektura prekritja. Uporabniško aplikacijo, napisano v jeziku HDL ali izdelano po postopku HLS prevedemo in sintetiziramo v jedra IP, ta pa se naložijo na čip FPGA.

Knjižnica PYNQ prekritjem doda vmesnik za programski jezik Python in gonilnike za pogosto uporabljana jedra IP. To nam omogoča, da implementirano logiko na čipu FPGA uporabimo kot navadno programsko knjižnico [10]. Za uporabo prekritja v PYNQ potrebujemo:

- datoteko `.bit`, v kateri je zapisana sintetizirana in implementirana digitalna logika, ki se bo naložila na FPGA. Ta logika je sestavljena iz enega ali več jeder IP,

- metapodatke o implementaciji v obliki datotek `.hwh` (angl. *hardware*

*handoff*) ter `.tcl` (angl. *tool command language*)[4] in

- ovojnico v višjenivojskem jeziku, ki zakrije podrobnosti komunikacije med procesorjem in čipom FPGA ter tako uporabniku prikaže preprost vmesnik za uporabo (klic funkcije).

Krovni razred za uporabo prekritij v PYNQ je razred je `Overlay`. Ta razred je zadolžen za nalaganje jeder IP iz datoteke `.bit` na čip FPGA ter za izbiro pravih gonilnikov za ta jedra. Gonilniki se uporabljajo za zakritje podrobnosti komunikacije med procesorjem in posameznim jedrom IP. Pri uporabi več jeder in zato tudi več gonilnikov lahko to logiko dodatno zakrijemo z implementacijo razreda, ki razširja razred `Overlay`.

Knjižnica PYNQ že vsebuje gonilnike za mnoge funkcionalnosti tako sistema na čipu Zynq kot tudi same razvojne plošče PYNQ:

- GPIO (LED, gumbi, stikala),
- DMA,
- zvok (integrirani mikrofon in zvočnik),
- sliko (HDMI in DisplayPort),
- nalaganje programov za mehki procesor Microblaze,
- vmesnike Raspberry Pi, Arduino, Pmod in Grove.

Razred `Overlay` izbere pravi gonilnik za jedro IP z uporabo metapodatka VLNV (angl. *vendor, library, name, and version*; proizvajalec, knjižnica, ime in različica), ki identificira vsako jedro IP. Gonilniki vsebujejo seznam VLNV vseh jeder, ki jih podpirajo. V primeru, da razred `Overlay` ne najde pravega gonilnika za jedro se uporabi gonilnik `DefaultIP`, ki omogoča le osnoven dostop do jedra IP (na primer dostop do registrov in pomnilniškega prostora). Pri razvoju lastnega gonilnika razširimo razred `DefaultIP` in dodamo metode, ki zakrijejo podrobnosti komunikacije.

[4]Datoteki `.hwh` in `.tcl` morata imeti enako ime kot datoteka `.bit`.

### 2.1.1 Primer uporabe ovojnice za prekritje

Kot primer uporabe knjižnice PYNQ vzamemo preprost pospeševalnik za seštevanje dveh števil in mu dodamo ovojnico PYNQ. Pospeševalnik vsebuje logiko seštevalnika ter logiko za komunikacijo s procesorjem[5]. To logiko zapakiramo v jedro IP ter ga v obliki datoteke `.bit` skupaj z datotekama `.hwh` in `.tcl` shranimo na razvojno ploščo PYNQ. Datoteki `.hwh` in `.tcl` vsebujeta med drugim podatke o:

- imenih jeder IP za seštevanje in komunikacijo s procesorjem,
- VLNV jeder IP,
- naslovih in imenih registrov, v katera moramo zapisati števili, ki jih bomo sešteli,
- naslovu in imenu registra, v katerem bo zapisan rezultat in
- frekvenci ure s katero deluje vezje.

Če logiko prekritja opišemo sami z uporabo okolja Vivado, nam to samodejno ustvari datoteki `.bit` in `.hwh`, datoteko `.tcl` pa naredimo z nekaj dodatnimi kliki[6].

Za jedro IP seštevalnika moramo napisati gonilnik, ki bo argumente zapisal v registre ter rezultat vrnil uporabniku. To uporabniku prikažemo kot preprost klic funkcije `sestej(a, b)`. Na zapisu 2.1 je prikazan primer implementacije takega gonilnika, na zapisu 2.2 pa uporaba tega gonilnika z privzetim razredom `Overlay`.

[5]Komunikacijo med procesorjem in čipom FPGA bomo opisali v naslednjih poglavjih.

[6]`File > Export > Export Block Design`.

```
from pynq import DefaultIP

class SestejDriver(DefaultIP):

    # Seznam VLNV imen za jedra, ki jih podpira ta gonilnik
    bindto = ['smole.org:user:adder:1.0']

    def __init__(self, description):
        super().__init__(description=description)

    # Za uporabnika pripravimo funkcijo sestej
    def sestej(self, a, b):
        self.register_map.a = a # Zapisemo prvi operand
        self.register_map.b = b # Zapisemo drugi operand
        return self.register_map.c  # Preberemo in vrnemo
   rezultat
```

Zapis 2.1: Gonilnik za seštevalnik.

```
from pynq import Overlay

overlay = Overlay('/home/xilinx/adder/adder.bit')

# Samodejno izbere nas gonilnik, ki ima funkcijo 'sestej'
rezultat = overlay.adder.sestej(20, 22)
print(rezultat)     # 42
```

Zapis 2.2: Uporaba gonilnika za seštevalnik.

Težava pri neposredni uporabi gonilnika je, da mora uporabnik še vedno poznati ime jedra IP, s katerim želi komunicirati (na zapisu 2.2 je to `adder` v predzadnji vrstici). Temu se izognemo tako, da implementiramo razred, ki razširja `Overlay` in v tem razredu napišemo metode, ki delovanje gonilnika dodatno zakrije uporabniku. Na ta način zakrijemo logiko tudi v primeru, da uporabljamo več kot eno jedro IP in s tem več kot en gonilnik ali pa imamo poleg klica gonilnika še dodatno logiko, ki je uporabnik ne rabi videti. Na zapisu 2.3 je prikazana implementacija razreda, ki razširja `Overlay`, na zapisu 2.4 pa uporaba tega prekritja.

```
from pynq import Overlay

class SestevalnikOverlay(Overlay):
    def __init__(self, bitfile, **kwargs):
        super().__init__(bitfile, **kwargs)

    def sestej(self, a, b):
        # Skrijemo klic funkcije iz gonilnika
        return self.adder.sestej(20, 22)
```

Zapis 2.3: Implementacija lastnega razreda za prekritje.

```
overlay = SestevalnikOverlay('/home/xilinx/adder/adder.bit')
rezultat = overlay.sestej(20, 22)
print(rezultat)     # 42
```

Zapis 2.4: Uporaba seštevalnika z lastnim razredom za prekritje.

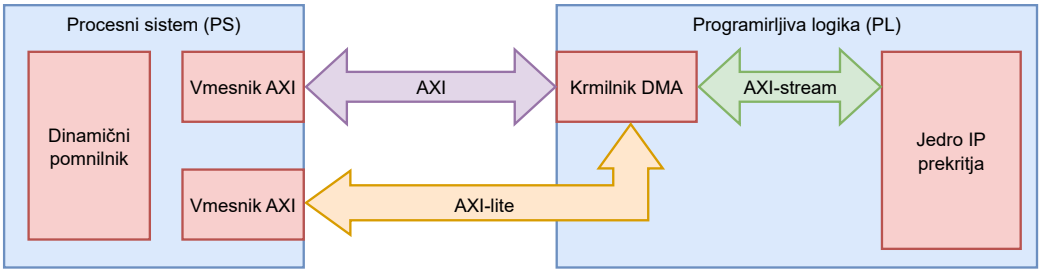
## 2.2 Komunikacija med procesorjem in FPGA

PYNQ podpira več različnih metod za komunikacijo med procesorjem, ki mu v kontekstu čipov SoC ponavadi rečemo procesni sistem (angl. *processing system*, PS) in čipom FPGA, ki mu rečemo programirljiva logika (angl. *programmable logic*, PL).

Najpreprostejši način je 64 povezav GPIO med PS in PL, preko katerih lahko pošiljamo enostavne kontrolne signale in signale za ponastavitev (angl. *reset*). Uporabimo jih lahko tudi za prekinitve, a za te pogosteje uporabimo namenske povezave. Prav tako nam je na voljo 9 vmesnikov AXI, ki jih lahko uporabimo z navadnim protokolom AXI, protokolom AXI-Lite ali protokolom AXI-Stream.

Za prenos podatkov v obliki dostopov do registrov (kot v primeru na zapisu 2.1) ponavadi uporabimo protokol AXI-Lite, za prenos velikih količin podatkov pa AXI-Stream. Pri tem lahko dodatno uporabimo DMA (angl. *direct memory access*) in s tem razbremenimo procesor. Krmilnika DMA nam ni potrebno pisati od začetka, saj lahko uporabimo Xilinxovo jedro IP [5].

Pri uporabi tega se za komunikacijo med PS in PL uporabljata AXI in AXI-Stream za prenos podatkov in AXI-Lite za nadzor nad komunikacijo, kot je prikazano na sliki 2.3.




Slika 2.3: Bločni diagram komunikacije z uporabo DMA. Vijolična povezava (AXI) krmilniku DMA omogoča dostop do dinamičnega pomnilnika, zeleno je označena povezava AXI-Stream, preko katere se prenašajo podatki med krmilnikom DMA in ostalimi jedri IP, oranžno pa je označena povezava AXI-Lite, ki se uporablja za nadzor krmilnika DMA.

### 2.2.1 Primer uporabe DMA

Knjižnica PYNQ že vsebuje gonilnik za jedro IP za krmilnik DMA ter kopico uporabnih funkcij za upravljanje s pomnilnikom. Ena izmed teh funkcij je `allocate`, s katero rezerviramo pomnilnik, iz katerega bo krmilnik DMA bral in/ali pisal. Funkciji podamo število elementov tabele in velikost elementa[7], ta pa za nas rezervira blok pomnilnika in nam vrne objekt, ki se obnaša kot tabela NumPy, poleg tega pa vsebuje še fizični naslov rezerviranega pomnilnika. Tega v ozadju uporablja krmilnik DMA pri branju ter pisanju. Na zapisu 2.5 je prikazana uporaba DMA s preprosto vrsto FIFO (angl. *first in, first out*).

Program naloži prekritje iz datoteke `fifo.bit`, pri čemer razred `Overlay` za uporabo jedra IP `axi_dma`, ki je del tega prekritja, samodejno izbere gonilnik za DMA. Pred uporabo gonilnika rezerviramo pomnilnik in vanj

[7]Podobno kot bi to storili s funkcijo `calloc` v jeziku C.

zapišemo podatke, ki jih bomo poslali. Na koncu začnemo prenos DMA ter počakamo, da se zaključi. Po koncu prenosa je vsebina tabel `input_buffer` in `output_buffer` enaka.

```
from pynq import allocate, Overlay

# Samodejno izbere pravi gonilnik
overlay = Overlay('fifo.bit')
dma = overlay.axi_dma

# Rezerviramo pomnilnik
input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)

# Zapisemo podatke
for i in range(5):
   input_buffer[i] = i

print(input_buffer) # 0 1 2 3 4

# Zacnemo prenos DMA
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)

# Pocakamo na konec prenosa
dma.sendchannel.wait()
dma.recvchannel.wait()

print(output_buffer) # 0 1 2 3 4
```

Zapis 2.5: Primer uporabe DMA iz jezika Python[8].

[8]Povzeto po `https://pynq.readthedocs.io/en/latest/pynq_libraries/dma.html` (pridobljeno 14. 6. 2023)

# Poglavje 3

# Protokol AXI

Protokol AXI (angl. *advanced extensible interface*) je razvilo podjetje ARM kot del specifikacije AMBA (*angl. advanced microcontroller bus architecture*). Omogoča hitro in preprosto povezovanje komponent v obliki arhitekture nadrejeni-podrejeni (angl. *master-slave*, včasih tudi *manager-suborbinate*) [2]. Nadrejena komponenta je komponenta, ki bo začela transakcijo (začela brati ali pisati), podrejena komponenta pa bo odgovarjala na to transakcijo.

Protokol AXI definira prenos podatkov med dvema točkama (angl. *point-to-point*). Če želimo med seboj povezati več komponent lahko uporabimo Xilinxovo jedro IP za AXI SmartConnect. Ta se obnašata podobno kot stikala pri omrežjih Ethernet in omogoča komunikacijo med več kot dvema komponentama. Poleg povezovanja več komponent omogoča tudi povezovanje komponent, ki implementirajo različne generacije ali tipe protokola AXI (na primer nadrejena komponenta implementira AXI4, podrejena pa AXI3[1] ali nadrejena implementira AXI, podrejena pa AXI-Lite) ter spreminjanje širine prenosa.

[1] AXI3 in AXI4 sta generaciji protokola AXI. Ob času pisanja je najpogosteje uporabljana AXI4, ki jo uporabljamo tudi v tej diplomski nalogi.

## 3.1 Kanali

Protokol AXI definira 5 kanalov, prek katerih si nadrejena in podrejena komponenta izmenjujeta podatke. Vsak kanal je sestavljen iz nekaj obveznih in množice neobveznih signalov, ki jih uporabimo glede na naše zahteve pri prenosu. Obvezni signali se uporabljajo za uskladitev prenosa, prenašanje naslovov in podatkov ter potrjevanje prejetja. Neobvezni signali se med drugim uporabljajo za določanje:

- toka podatkov, če jih preko enega kanala prenašamo več (na primer preko enega kanala pošiljamo več slik hkrati),
- kvalitete storitve (angl. *quality of service*), ki določi prioriteto prenosa,
- zaklepanja in zaščite, s čimer si zagotovimo atomične operacije tudi če lahko do istega pomnilnika hkrati dostopa več komponent,
- vrstnega reda podatkov, če jih pošiljamo neurejene,
- zadnjega kosa podatkov znotraj transakcije.

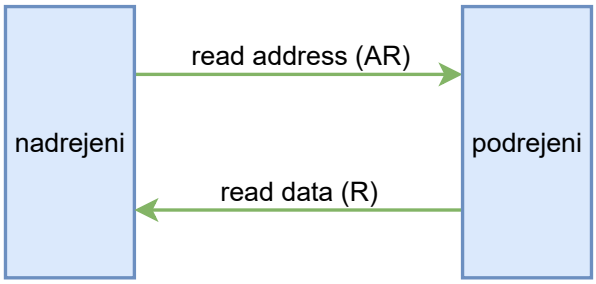
### 3.1.1 Branje

Za prenos podatkov od podrejenega k nadrejenemu (branje), ki je prikazano na sliki 3.1, se uporabljata kanala:

- read address (AR), preko katerega nadrejeni pošlje naslov, iz katerega želi brati in
- read data (R), preko katerega podrejeni pošlje podatke na zahtevanem naslovu.

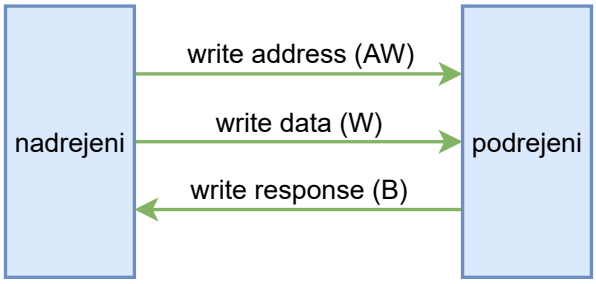
### 3.1.2 Pisanje

Za prenos podatkov od nadrejenega k podrejenemu (pisanje), ki je prikazano na sliki 3.2, se uporabljajo kanali:




Slika 3.1: Kanala, potrebna za prenos podatkov od podrejenega k nadrejenemu (branje).

- write address (AW) preko katerega nadrejeni podrejenemu pošlje naslov, na katerega želi pisati,
- write data (W), preko katerega nadrejeni podrejenemu pošlje podatke in
- write response (B), preko katerega podrejeni po koncu prenosa nadrejenemu sporoči stanje prenosa.




Slika 3.2: Kanali, potrebni za prenos podatkov od nadrejenega k podrejenemu (pisanje).

Ker so kanali za branje in pisanje ločeni, lahko komponenta hkrati bere in piše in s tem pospeši prenos podatkov.

## 3.2 Uskladitev prenosa

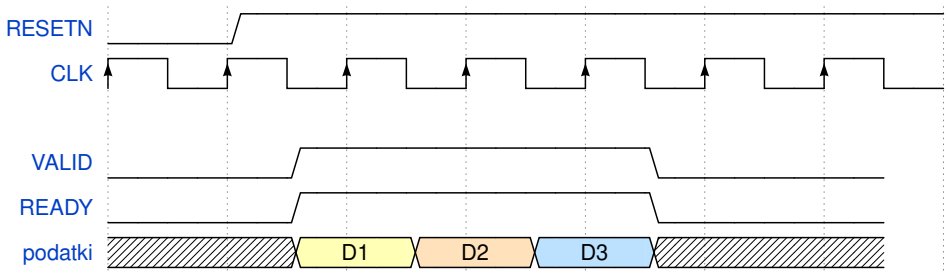
Vsak od petih kanalov mora pred prenosom podatkov uskladiti nadrejeno in podrejeno komponento. Za uskladitev prenosa (angl. *handshake*) vsak kanal uporablja svoja signala `VALID` in `READY`. Ponavadi imenu signala kot predpono dodamo ime kanala. Tako se na primer signal `VALID` v kanalu write data (W) imenuje `WVALID`, v kanalu write address (AW) pa `AWVALID`.

Signal `VALID` postavi nadrejeni (pošiljatelj), ko so podatki pripravljeni za pošiljanje, signal `READY` pa postavi podrejeni (prejemnik), ko je pripravljen za sprejetje podatkov. Prenos se zgodi le ob pozitivni urini fronti, ko sta oba signala postavljena na logično 1. Na slikah 3.3, 3.4 in 3.5 so prikazana vsa tri stanja signalov `VALID` in `READY`, preden se lahko zgodi prenos.

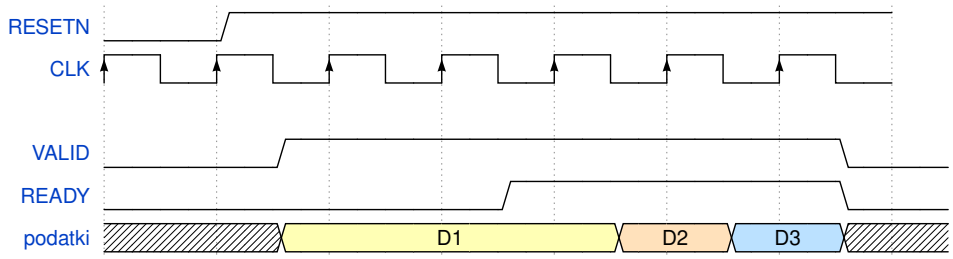
Signal `podatki` predstavlja katerikoli signal, ki ga uporabljamo pri prenosu podatkov. To je lahko na primer naslov za branje ali pisanje, podatki, ki jih želimo poslati itd.

Ker hitrost prenosa podatkov nadzirata tako pošiljatelj kot sprejemnik, lahko v primeru prehitrega prenosa svoj signal postavita na logično ničlo in s tem začasno ustavita prenos, nato spet začneta ko so prejšnji podatki obdelani.

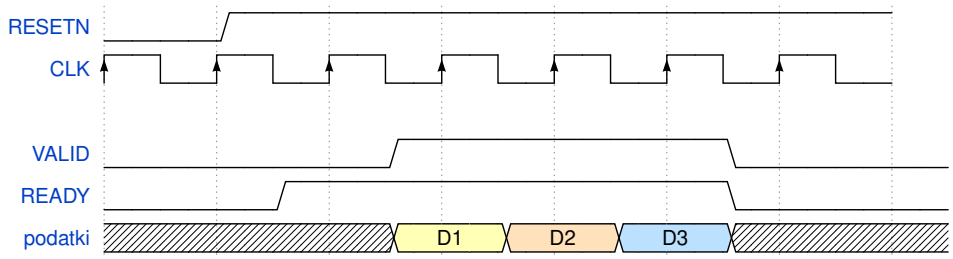
Za ponastavitev se uporablja signal `RESETN`, ki je aktiven v nizkem stanju.




Slika 3.3: Signala `VALID` in `READY` sta hkrati postavljena na logično 1.




Slika 3.4: Signal `VALID` je postavljen na logično 1 pred signalom `READY`. Prenos se zgodi šele, ko je na logično 1 postavljen tudi signal `READY`.




Slika 3.5: Signal `READY` je postavljen na logično 1 pred signalom `VALID`. Prenos se zgodi šele, ko je na logično 1 postavljen tudi signal `VALID`.

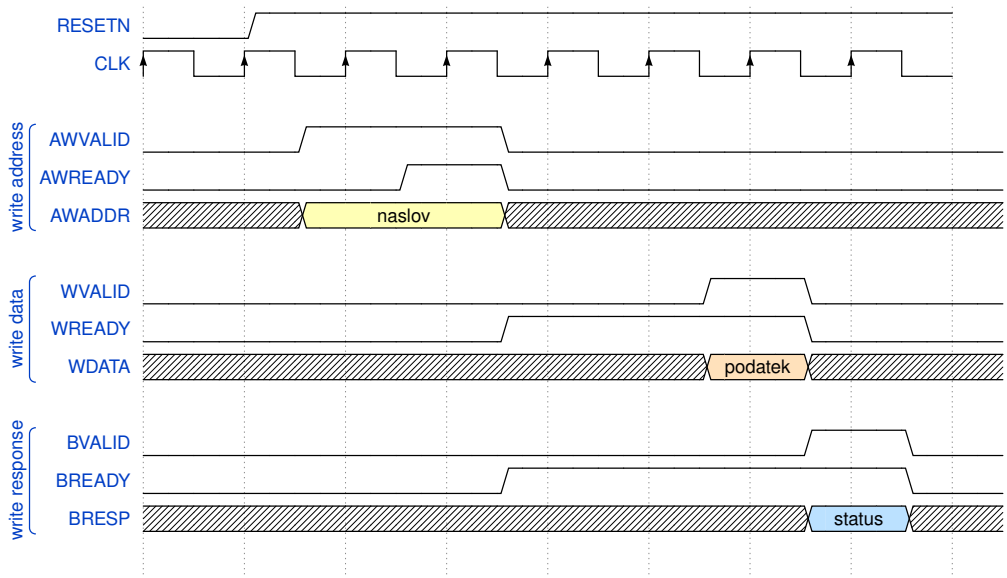
## 3.3 AXI-Lite

Protokol AXI definira veliko signalov in funkcionalnosti, ki jih v praksi pogosto ne potrebujemo. Primer takih funkcionalnosti sta na primer rafalni prenos (zaporedni prenos več besed v enem paketu, kot je prikazano na slikah 3.3, 3.4 in 3.5) in zelo široka vodila (AXI dovoli širine od 32 do 1024 bitov).

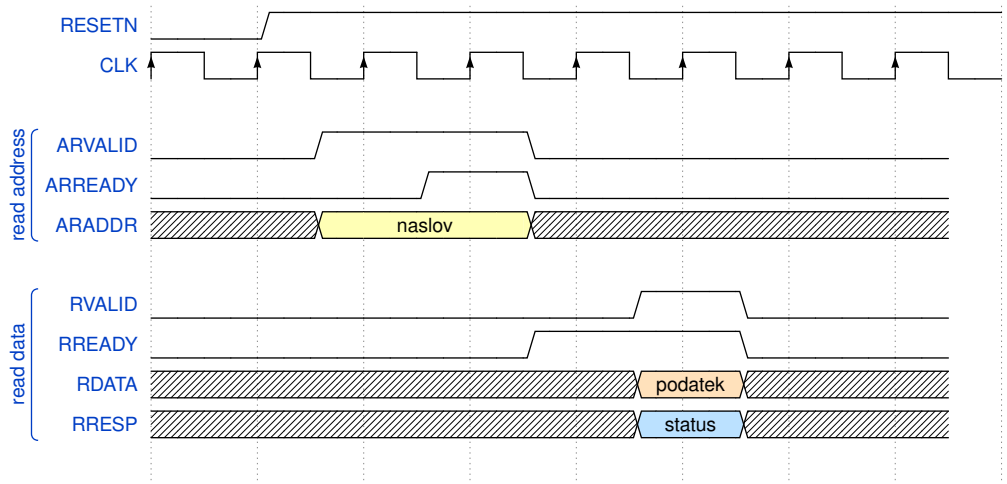
V primeru da želimo protokol AXI uporabiti za prenos manjše količine podatkov v obliki branj in pisanj na določen naslov (dostop do registrov), lahko namesto protokola AXI uporabimo protokol AXI-Lite, ki zgoraj naštetih, in tudi kup drugih, funkcionalnosti ne podpira. To nam omogoča načrtovanje preprostejših implementacij, ki zasedejo manj virov. Širina prenosa pri AXI-Lite (širina signalov `WDATA` in `RDATA`) je 32 ali 64 bitov.

Slika 3.6 prikazuje pisanje s protokolom AXI-Lite, slika 3.7 pa branje s

protokolom AXI-Lite.




Slika 3.6: Pisanje s protokolom AXI-Lite. Najprej na kanalu write address pošljemo naslov, na katerega pišemo, nato na kanalu write data pošljemo podatek in na koncu na kanalu write response dobimo status prenosa.



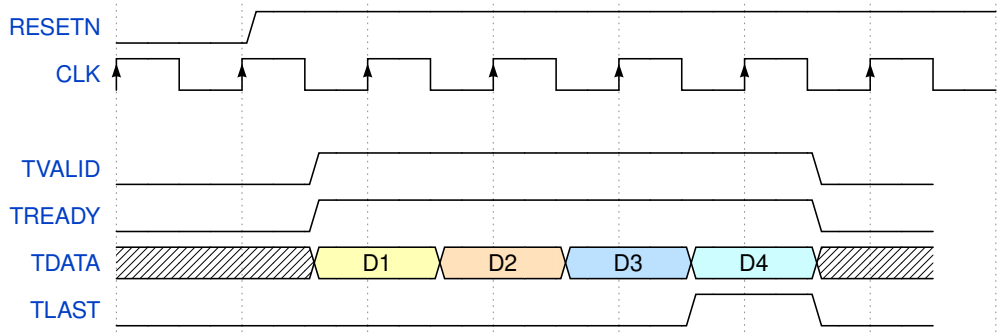

Slika 3.7: Branje s protokolom AXI-Lite. Najprej na kanalu read address pošljemo naslov, iz katerega želimo brati, nato pa na kanalu read data dobimo zahtevani podatek in status prenosa.

## 3.4 AXI-Stream

Za prenos velikih količin podatkov v eno smer, na primer pri uporabi DMA, lahko uporabimo protokol AXI-Stream (včasih poimenovan tudi AXIS). Ta namesto petih kanalov uporablja le enega. Ne skrbi za signaliziranje naslovov in statusa prenosa, temveč le za čim hitrejši in čim preprostejši prenos podatkov [1].

Za delovanje poleg signalov za uro in ponastavitev potrebuje le še signale `TVALID`, `TREADY` in `TDATA`[2]. V primeru da uporabljamo Xilinxov krmilnik DMA, moramo tem signalom dodati še signal `TLAST`, ki signalizira konec prenosa.

Širina prenosa (širina signala `TDATA`) je lahko 32, 64, 128, 256, 512 ali 1024 bitov. Na sliki 3.8 je prikazano pošiljanje paketa, sestavljenega iz 4 besed s protokolom AXI-Stream.




Slika 3.8: Časovni diagram za pošiljanje paketa, sestavljenega iz 4 besed s protokolom AXI-Stream.

[2]AXI-Stream kot predpono imen signalov uporablja črko 'T'.

# Poglavje 4

# Algoritem MD5

MD5 (angl. *message digest 5*) je algoritem za zgoščevanje poljubne količine podatkov v 128 bitno zgoščeno vrednost. Algoritem je leta 1992 razvil Ronald Rivest kot nadomestilo za algoritem MD4 [15]. V preteklosti se je zelo pogosto uporabljal v kriptografiji kot zgoščevalna funkcija pri shranjevanju gesel. Dandanes se za to ne uporablja več, saj je trke[1] možno najti v nekaj sekundah [13]. Kljub temu se še vedno pogosto uporablja za potrditev pravilnosti prenosa podatkov, ugotavljanje enakosti datotek in podobno.

## 4.1 Opis algoritma

Algoritem za poljubno dolg vhod izračuna 128-bitno zgoščeno vrednost [12]. Zaporedju bitov vhodnih podatkov na konec doda bitno enico in toliko ničel, da je skupna dolžina vhodnih podatkov v bitih po modulu 512 enaka 448. Te podatke razdeli na 512-bitne bloke, pri čemer bo zadnji blok dolg le 448 bitov. V 64 bitov, ki mu manjkajo do dolžine 512 bitov zapiše dolžino prvotnih vhodnih podatkov po pravilu tankega konca.

Bloke enega za drugim zgoščuje tako, da vsak blok najprej razdeli na 16 32-bitnih besed, nato pa nad temi besedami izvrši 64 iteracij logičnih funkcij, ki so sestavljene iz operatorjev AND, OR, NOT in XOR. Prvih 16 iteracij se

[1] Več vhodov, ki se preslikajo v enak izhod (v enako zgoščeno vrednost).

izvaja stopnja (angl. *round*) `F`, naslednjih 16 stopnja `G`, nato `H` in na koncu `I`. Stopnje se med seboj razlikujejo v logičnih funkcijah, ki jih izvedejo nad vhodnimi besedami. Psevdokoda algoritma je zapisana v zapisu 4.1.

```
fun MD5(data):
    int S[64], K[64]    # Tabeli konstant
    int a0 = 0x67452301
    int b0 = 0xefcdab89
    int c0 = 0x98badcfe
    int d0 = 0x10325476

    size = len(data)
    data += '1' # Dodaj '1'
    while len(data) % 512 != 448
        data += '0' # Dodajaj '0'
    data += size # Na konec dodaj dolzino

    # Podatke razdeli na 512-bitne bloke
    C[] = split(data, 512)
    for i in 0..length(C) - 1:
        # Blok razdeli na 32-bitne besede
        W = split(C[i], 32)
        int a, b, c, d = a0, b0, c0, d0

        for i from 0 to 63:
            int f, g
            if 0 <= i <= 15:        # Stopnja F
                f = (b and c) or (not b and d)
                g = i
            else if 16 <= i <= 31:  # Stopnja G
                f = (d and b) or (not d and c)
                g = (5*i + 1) mod 16
            else if 32 <= i <= 47:  # Stopnja H
                f = b xor c xor d
                g = (3*i + 5) mod 16
            else if 48 <= i <= 63:  # Stopnja I
                f = C xor (b or not d)
                g = (7*i) mod 16
```

```
            temp = b + rotate_left(a + f + K[i] + W[g], s[i])
            a = d; d = c; c = b; b = temp

        a0 += a; b0 += b; c0 += c; d0 += d

return concat(a0, b0, c0, d0)
```

Zapis 4.1: Psevdokoda algoritma MD5[2].

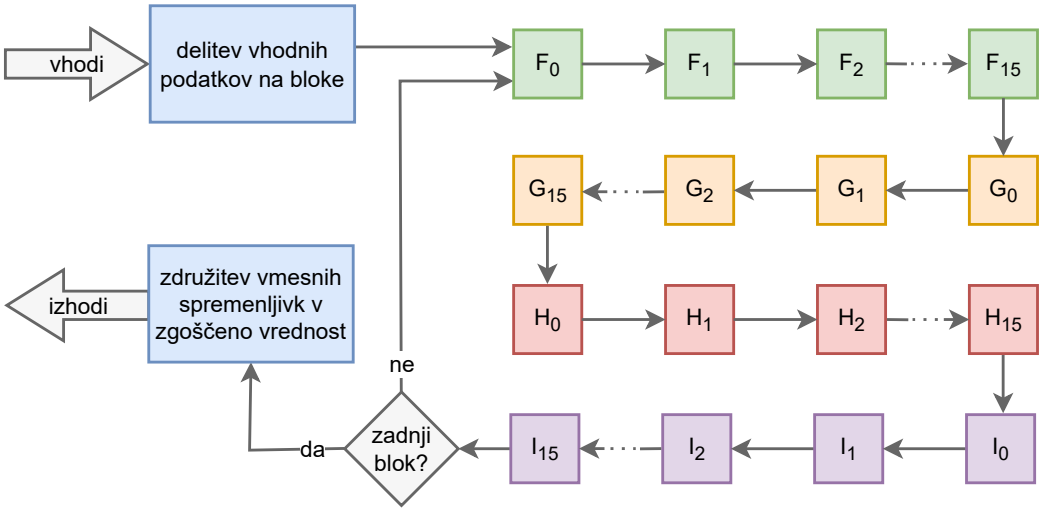
Realizacija zgoščevalnih in šifrirnih algoritmov v vezju FPGA je v mnogih primerih veliko hitrejša in učinkovitejša kot z uporabo CPE, saj taki algoritmi podatke pogosto obdelujejo na nivoju bitov ali bajtov, pri tem pa uporabljajo preproste logične funkcije, ki jih je na vezjih FPGA zelo preprosto opisati in izvajati [14, 6].

## 4.2 Cevovodni algoritem

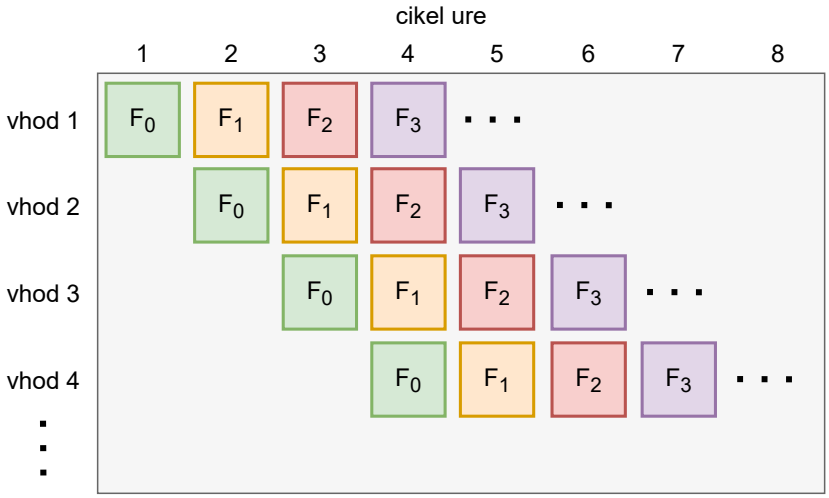
Algoritem MD5 nad vsakim blokom podatkov izvede 64 iteracij logičnih funkcij. Iteracije so odvisne le od izhoda prejšnje iteracije, kar pomeni, da lahko algoritem obdelavo enega bloka izdelamo v obliki 64-stopenjskega cevovoda, ki je prikazan na sliki 4.1. Na žalost za izračun zgoščene vrednosti bloka podatkov potrebujemo vrednost vmesnih spremenljivk, ki so rezultat zgoščevanja prejšnjega bloka, kar pomeni, da cevovoda ne moremo uporabiti za pospešitev izračuna zgoščene vrednosti enega vhoda. Ta lastnost mnogih zgoščevalnih algoritmov zelo oteži izdelavo implementacij, ki podatke zgoščujejo vzporedno.

To omejitev lahko omilimo, če predpostavimo, da bomo algoritem uporabljali za izračun zgoščenih vrednosti za več vhodnih podatkov (na primer za več datotek). V tem primeru lahko v cevovodu hkrati računamo zgoščene vrednosti do 64 vhodov, kot je prikazano na sliki 4.2.

[2]Povzeto po `https://en.wikipedia.org/wiki/MD5` (pridobljeno 9. 6. 2023)




Slika 4.1: Algoritem za izračun zgoščene vrednosti MD5 s 64-stopenjskim cevovodom.




Slika 4.2: Uporaba cevovoda za izračun zgoščene vrednosti za več vhodov hkrati.

# Poglavje 5

# Izdelava pospeševalnika

Pospeševalnik smo razvili iz treh delov:

- jedra IP, ki zgoščuje podatke in izračuna zgoščeno vrednost,
- jedra IP, ki bo povezalo jedra za SoC, DMA in izračun zgoščene vrednosti ter
- ovojnice za prekritje v okolju Juypter.

Jedro za zgoščevanje podatkov smo opisali v jeziku VHDL (angl. *very high speed integrated circuit hardware description language*). Od krmilnika DMA sprejema podatke, izračuna zgoščeno vrednost in jo vrne krmilniku DMA. Za komunikacijo s krmilnikom DMA uporablja protokol AXI-Stream.

Izdelano in zapakirano jedro za izračun zgoščenih vrednosti smo vključili v drugo jedro IP, ki mu v tej diplomski nalogi rečemo glavno jedro. V glavnem jedru smo povezali uporabljana jedra IP povezali in nastavili njihove parametre (na primer širino podatkovnih vodil). Tu smo določili tudi frekvenco ure.

Dokončano glavno jedro IP smo sintetizirali in implementirali ter ustvarjeno prekritje v obliki datotek `.bit`, `.tcl` in `.hwh` kopirali na razvojno ploščo PYNQ-ZU. Na razvojni plošči smo v okolju Jupyter napisali ovojnico za to prekritje. Namen ovojnice je pripraviti podatke za obdelavo na čipu FPGA

in uporabniku prikazati prijazen vmesnik za uporabo v obliki preprostega klica funkcije.

## 5.1 Glavno jedro IP

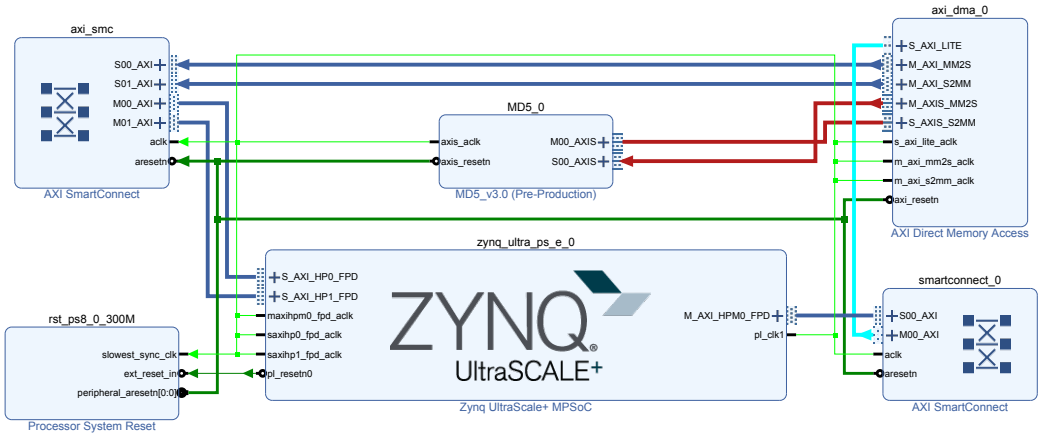
Glavno jedro IP, ki je prikazano na sliki 5.1, vsebuje jedra IP za:

- Zynq MPSoC (`Zynq UltraScale+ MPSoC`),
- krmilnik DMA (`AXI Direct Memory Access`),
- izračun MD5 (`MD5_v3.0`),
- povezovanje komponent, ki podpirajo drugačne različice protokola AXI (`AXI SmartConnect`),
- nadzor ponastavitenega signala (`Processor System Reset`).

Povezave so obarvane glede na njihov namen oziroma tip:

- svetlo zelena – urin signal,
- temno zelena – ponastavitneni signal,
- temno modra – AXI,
- svetlo modra – AXI-Lite,
- rdeča – AXI-Stream.

V jedru Zynq MPSoC (na sliki 5.1 spodaj na sredini) nastavimo uro in potrebne vmesnike AXI. Potrebujemo 3 vmesnike AXI: dva za prenos podatkov od in do krmilnika DMA (`S_AXI_HP0_FPD` in `S_AXI_HP1_FPD`) ter enega za nadzor krmilnika DMA (`M_AXI_HPM0_FPD`). Slednji se z uporabo jedra AXI SmartConnect spremeni v AXI-Lite, ki ga podpira krmilnik DMA. Signala za prenos podatkov pa omogočata krmilniku DMA pomnilniško preslikan dostop do dinamičnega pomnilnika.




Slika 5.1: Bločni diagram glavnega jedra IP.

Krmilnik DMA (na sliki 5.1 zgoraj desno) ima preko signalov `M_AXI_MM2S`[1] in `M_AXI_S2MM`[2] dostop do dinamičnega pomnilnika. Podatke, prebrane iz dinamičnega pomnilnika, preko vmesnika AXI-Stream `M_AXIS_MM2S` pošlje do jedra za izračun MD5, preko vmesnika `S_AXIS_S2MM` pa prejme izračunane zgoščene vrednosti in jih zapiše nazaj v dinamični pomnilnik.

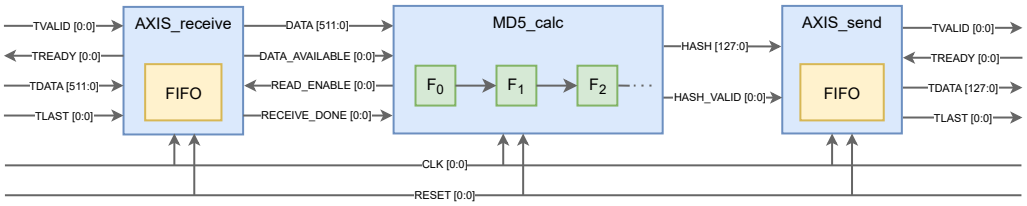
Jedro za nadzor ponastavitvenega signala (na sliki 5.1 spodaj levo) sinhronizira ponastavitveni signal z uro, določi njegovo trajanje in sproži ponastavitev ob zagonu (angl. *power-on reset*).

## 5.2 Jedro IP za izračun zgoščene vrednosti

Jedro IP za izračun zgoščene vrednosti z algoritmom MD5 je sestavljeno iz treh komponent: sprejemanja podatkov (`AXIS_receive`), izračuna zgoščenih vrednosti (`MD5_calc`) in pošiljanja zgoščenih vrednosti (`AXIS_send`). Slika 5.2 prikazuje diagram tega jedra IP skupaj s signali, ki povezujejo komponente.

[1] MM2S je angl. *memory-mapped to stream*

[2] S2MM je angl. *stream to memory-mapped*




Slika 5.2: Diagram jedra IP za izračun zgoščene vrednosti MD5 s signali, ki povezujejo komponente.

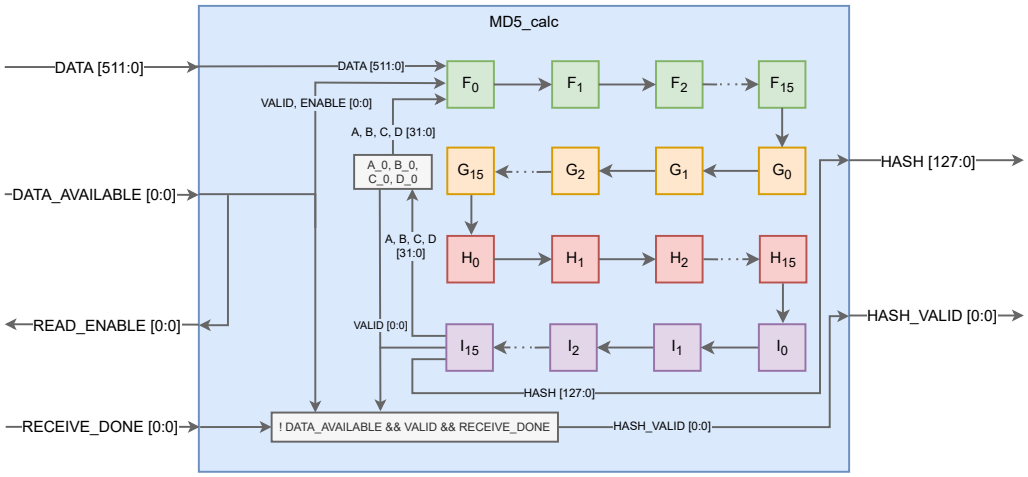
### 5.2.1 Sprejem podatkov

Za sprejemanje podatkov je zadolžena AXI-Stream podrejena (angl. *slave/suborbinate*) komponenta `AXIS_receive`. Ta od nadrejene komponente (krmilnika DMA) sprejema podatke in jih zapisuje v vrsto FIFO. Namen vrste FIFO je začasno shraniti prejete podatke preden jih zahteva komponenta za izračun zgoščene vrednosti. Za signaliziranje, da so podatki na voljo, se uporablja signal `DATA_AVAILABLE`. V primeru, da je prejemanje podatkov počasnejše kot računanje zgoščenih vrednosti, se signal `DATA_AVAILABLE` postavi na logično 0 in s tem sporoči komponenti `MD5_calc`, da mora ustaviti cevovod in počakati na nove podatke. Signala `READ_ENABLE` in `DATA` se uporabljata za branje iz vrste FIFO, signal `RECEIVE_DONE` pa pove, da je komponenta od krmilnika DMA prejela signal `TLAST` in zato ne pričakuje novih podatkov.

### 5.2.2 Izračun zgoščenih vrednosti

Izračun zgoščenih vrednosti z algoritmom MD5 izvaja komponenta `MD5_calc`. Ta vsebuje 64-stopenjski cevovodni algoritmom, ki je opisan v poglavju 4.2. Diagram na sliki 5.3 prikazuje ključne dele realizacije tega algoritma v vezju FPGA.

Postavitev signala `DATA_AVAILABLE` na logično 1 komponenti `MD5_calc` sporoči, da lahko začne zgoščevati podatke, ki jih prebere iz vrste FIFO komponente `AXIS_receive` s signalom `READ_ENABLE`. Ta signal lahko povežemo kar na signal `DATA_AVAILABLE`, saj bomo tako brali natanko tedaj, ko so po-




Slika 5.3: Komponenta za izračun zgoščenih vrednosti na FPGA.

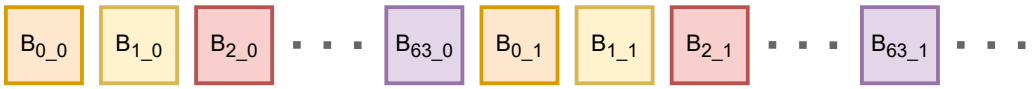
datki na voljo. Poleg tega lahko signal `DATA_AVAILABLE` uporabimo tudi za nadzor delovanja cevovoda, saj se mora ta v primeru pomanjkanja podatkov ustaviti (signal `ENABLE`), hkrati pa nam pove tudi, da je vrednost signala `DATA` veljavna (signal `VALID`). Postopek zgoščevanja podatkov iz vrste FIFO ponavljamo, dokler ta vsebuje podatke in ne prejmemo signala `RECEIVE_DONE`. Ko vrsto izpraznimo in je signal `RECEIVE_DONE` postavljen na logično 1, je izhod cevovoda končni rezultat zgoščevanja in ga lahko povežemo na izhod komponente.

V cevovod poleg signala za uro in ponastavitev povežemo signale:

- `DATA`, ki vsebuje 512-bitne bloke podatkov,
- `VALID`, ki pove, da je vrednost signala `DATA` veljavna,
- `ENABLE`, ki nadzira delovanje cevovoda,
- `A`, `B`, `C` in `D`, ki vsebujejo začetne konstante (`A_0`, `B_0`, `C_0` in `D_0`) ali rezultat zgoščevanja prejšnjega bloka.

Cevovod pričakuje bloke podatkov razvrščene tako, da se izmenjujejo bloki $B_{i\_j}$, kjer je $i \in \{0..63\}$ indeks vhoda in $j \geq 0$ zaporedna številka

bloka za vsakega od 64 različnih vhodov, kot je prikazano na sliki 5.4. Zaradi preprostejše izdelave smo predobdelavo podatkov in razvrščanje blokov napisali v ovojnici prekritja v okolju Jupyter.




Slika 5.4: Vrstni red blokov podatkov. Najprej prvi blok prvega vhoda, nato prvi blok drugega vhoda, nato tretjega do 64-tega vhoda. Temu sledi drugi blok prvega vhoda, drugi blok drugega vhoda in tako naprej do konca vhodnih podatkov.

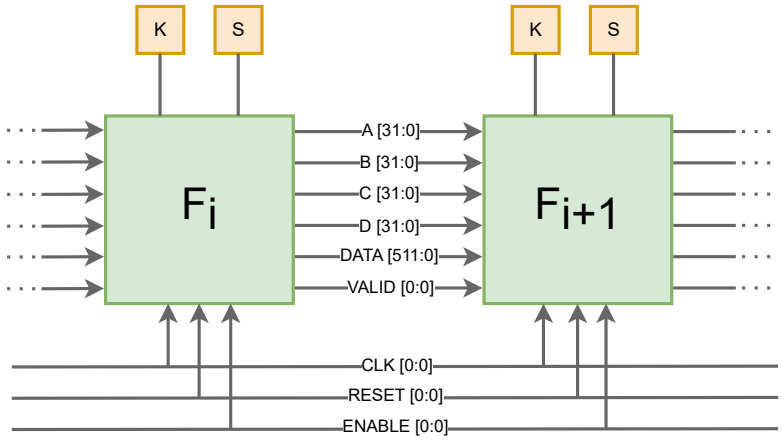
Vsaka od 64 stopenj cevovoda je povezana na uro, ponastavitveni signal ter signal za nadzor delovanja cevovoda. Med iteracijami se pošiljajo:

- vrednosti 32-bitnih spremenljivk `A`, `B`, `C` in `D`,
- 512-bitni blok podatkov, ki se v trenutno zgoščuje `DATA`,
- zastavico, ki signalizira veljavnost podatkov (`VALID`), saj cevovod prvih 64 urnih period (preden se napolni) vsebuje neveljavne podatke.

Prejete vrednosti mora vsaka stopnja po koncu izračuna poslati naslednji stopnji v cevovodu, kot je prikazano na sliki 5.5. Poleg zgoraj naštetih signalov ima vsaka stopnja dostop tudi do lastne kopije tabel konstant K in S. Tabeli izdelamo kot preprost bralni pomnilnik (angl. *read-only memory*, ROM).

### 5.2.3 Pošiljanje izračunanih zgoščenih vrednosti

Za pošiljanje izračunanih zgoščenih vrednosti je zadolžena nadrejena (angl. *master/manager*) komponenta `AXIS_send`. Ta podrejeni komponenti (krmilniku DMA) pošilja izračunane zgoščene vrednosti, ki so začasno shranjene v vrsti FIFO, saj se mora v primeru prehitrega pošiljanja ustaviti in počakati, da je DMA krmilnik spet pripravljen. Zgoščene vrednosti v vrsto vstavlja komponenta `MD5_calc` z uporabo signalov `HASH` in `HASH_VALID`.




Slika 5.5: Signali med iteracijami.

## 5.3 Ovojnica za prekritje v PYNQ

V okolju Jupyter želimo uporabniku ponuditi kar se da preprost vmesnik za naš program, zato smo napisali ovojnico za prekritje, ki uporabniku prikaže le funkcijo `md5(podatki)`, ki izračuna zgoščene vrednosti z algoritmom MD5. V ozadju moramo te podatke pripraviti za zgoščevanje na čipu FPGA. Končni rezultat priprave so 512-bitni bloki podatkov, ki so razvrščeni, kot je prikazano na sliki 5.4.

Programski jezik Python zaradi njegove zasnove ne omogoča hitrega izvajanja, zato smo to pripravo podatkov izvedli v programskem jeziku C. Povezavo med jezikoma smo dosegli z uporabo knjižnice CFFI (angl. *C Foreign Function Interface*), ki je že naložena na razvojne plošče PYNQ. Ta knjižnica omogoča klic funkcij, napisanih v programskem jeziku C, iz programskega jezika Python. Zapis 5.1 prikazuje uporabo te knjižnice. V datoteki `preprocess.c` smo napisali kodo za pripravo podatkov. Vhodna točka v to kodo je funkcija `preprocess`, katere prototip smo podali tudi knjižnici CFFI, saj bo ta tako vedela kako funkcijo pognati. Funkciji kot parametre podamo kazalec na pomnilnik, kamor bo funkcija zapisala rezultat, kazalec na tabelo vhodov ter kazalec na tabelo dolžin vhodov. Knjižnica CFFI bo kodo v programskem jeziku C samodejno prevedla in pripravila za uporabo.

```
from cffi import FFI

ffi = FFI()

with open("preprocess.c") as f:
    c_code = f.read()

# Prototip funkcije, ki jo klicemo iz jezika Python
ffi.cdef("""
    void preprocess(unsigned char * buffer_out,
                    unsigned char **inputs,
                    int *sizes);
    """)
C = ffi.verify(c_code) # Prevede kodo
```

Zapis 5.1: Priprava knjižnice CFFI za uporabo iz programskega jezika Python.

### 5.3.1 Predobdelava za MD5

Vsak vhod moramo pred zgoščevanjem najprej obdelati. V jeziku C to storimo tako, da rezerviramo pomnilnik za obdelane podatke z uporabo funkcije `calloc` ter vanj kopiramo vhodne podatke. Tem podatkom dodamo bit '1', oziroma bajt '0x80', saj bo naša implementacija delala na nivoju bajtov. Za dodajanje ničel nam ni potrebno skrbeti, saj je za to poskrbela funkcija `calloc`. Na konec moramo le še dodati prvotno dolžino podatkov po pravilu tankega konca. Koda za predobdelavo podatkov je prikazana na zapisu 5.2. Funkcija prejme kazalec na tabelo vseh vhodov, kazalec na tabelo dolžin teh vhodov ter indeks vhoda, ki se trenutno obdeluje.

```
void preprocess_input(unsigned char **data_arrays,
                      int* sizes, int index) {

    unsigned char *data_arr = data_arrays[index];
    int size = sizes[index];
```

```
    // Izracunaj novo velikost tabele za podatke
    int new_size = (((size+9) + 63) / 64) * 64;

    // Rezerviraj prostor, calloc ga popise z niclami
    unsigned char *new_data_arr = (unsigned char *)calloc(
   new_size, sizeof(unsigned char));

    // Kopiraj podatke v novo tabelo
    memcpy(new_data_arr, data_arr, size);

    // Popravi kazalec na tabelo in velikost tabele
    data_arrays[index] = new_data_arr;
    sizes[index] = new_size;

    // Na konec doda bit '1'
    new_data_arr[size] = 0x80;

    // Na konec dodaj dolzino prvotnih podatkov
    long original_length = size * 8;
    for(int i = 0; i < 8; i++)
        new_data_arr[new_size - 8 + i] = (original_length >>
   (i * 8) & 0xff);
}
```

Zapis 5.2: Funkcija za predobdelava vsakega od vhodov.

### 5.3.2 Razvrstitev blokov za cevovod

Cevovod na čipu FPGA pričakuje, da bodo bloki podatkov vseh 64 vhodov prepleteni kot je prikazano na sliki 5.4. To dosežemo s kodo na zapisu 5.3, ki prejme kazalec na medpomnilnik za rezultat, kazalec na tabelo vhodov in kazalec na tabelo dolžin teh vhodov. Prejete podatke s kopiranjem v medpomnilnik za rezultat pravilno razvrsti. Izhod funkcije je zaporedje bajtov, ki ga lahko beremo kot zaporedje 512-bitnih blokov podatkov in ga zato lahko pošljemo na zgoščevanje na čipu FPGA.

```
void interleave(unsigned char *output_buffer, unsigned char
    **data_arrays, int* sizes) {

     int pos = 0;
     for (int block_index = 0; block_index < sizes[0] / 64;
    block_index++) {
          for (int array_index = 0; array_index < 64;
    array_index++) {
               memcpy(output_buffer + pos, data_arrays[
    array_index] + (block_index * 64), 64 * sizeof(char));
               pos += 64;
          }
     }
}
```

Zapis 5.3: Prepletanje blokov podatkov.

### 5.3.3 Izdelava ovojnice

Pravilno razvrščene bloke podatkov z uporabo gonilnika za DMA kopiramo iz dinamičnega pomnilnika na čip FPGA, kjer se v cevovodu zgostijo in prenesejo nazaj v dinamični pomnilnik. Koda celotne ovojnice za prekritje, ki izračuna zgoščene vrednosti z algoritmom MD5, je prikazana na zapisu 5.4.

```
class MD5Overlay(Overlay):
     def __init__(self, bitfile, **kwargs):
          super().__init__(bitfile, **kwargs)

     def md5(self, data_list):

          # Pripravi kazalce na vhode in na velikosti teh
    vhodov
          data_list_ptrs = ffi.new("unsigned char*[]", [ffi.
    from_buffer(l) for l in data_list])
          sizes = [len(data_list[0])]*64
          sizes_ptr = ffi.new("int[]", sizes)

          # Pripravi prostor v katerega bo C zapisal rezultat
```

```
        size = (((sizes[0]+9) + 63) // 64) * 64 * 64
        buffer_out = ffi.new("unsigned char[]", size)

        # Klici C program
        C.preprocess(buffer_out, data_list_ptrs, sizes_ptr)
        interleaved = np.frombuffer(ffi.buffer(buffer_out,
    size), dtype=uint512)

        # Pripravi pomnilnik za DMA
        input_buffer = allocate(shape=(np.shape(interleaved)
    [0],), dtype=uint512)
        output_buffer = allocate(shape=(64,), dtype=uint128)
        np.copyto(input_buffer, interleaved)

        # Prenesi na FPGA in nazaj
        dma = self.axi_dma_0
        dma.sendchannel.transfer(input_buffer)
        dma.recvchannel.transfer(output_buffer)
        dma.sendchannel.wait()
        dma.recvchannel.wait()

        return output_buffer
```

Zapis 5.4: Ovojnica prekritja za izračun zgoščenih vrednosti z algoritmom MD5.

Napisana ovojnica nam omogoča uporabo pospeševalnika za izračun zgoščenih vrednosti z algoritmom MD5, kot je prikazano na zapisu 5.5. Funkcija kot vhod pričakuje seznam 64 vhodov tipa `bytearray`. Če želimo dodati podporo tudi za drugačne tipe podatkov (na primer besedilo, številke, datoteke itd.) moramo v funkciji `md5` dodati še logiko za pretvorbo v `bytearray`.

```
overlay = MD5Overlay('/home/xilinx/overlays/md5.bit')
overlay.md5([b'abcd']*64)
```

Zapis 5.5: Uporaba ovojnice za izračun zgoščenih vrednosti z algoritmom MD5.

# Poglavje 6

# Rezultati

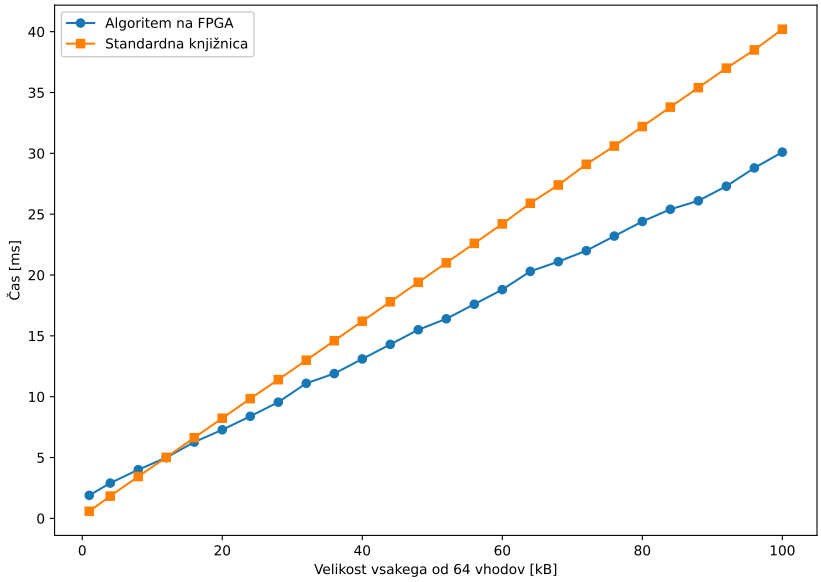
Našo rešitev smo primerjali s funkcijo `hashlib.md5()`, ki je del Pythonove standardne knjižnice in je napisana v jeziku C. Vse meritve smo izvedli na razvojni plošči PYNQ-ZU, torej na procesorju ARM Cortex-A53 in vgrajenemu čipu FPGA. Standardna knjižnica se je izvajala le na procesorju ARM, medtem ko je naš pospeševalnik procesor ARM uporabljal za predpripravo podatkov, zgoščeval pa na čipu FPGA. Logika na čipu FPGA je delovala s frekvenco ure 150 MHz.

Hitrost algoritmov smo izmerili s pomočjo orodja `timeit`, ki ga v okolju Jupyter uporabimo z oznako `%%timeit` na začetku celice. Orodje nato večkrat izvede vsebino celice ter nam izračuna povprečno trajanje in standardno napako. Da so meritve kar se da natančne, orodje vsebino celice najprej $n$-krat izvede v zanki, nato pa to ponovi $r$-krat, iz česar lahko izračuna napako. Vsebino celice skupno izvede $(n \cdot r)$-krat. V našem primeru je bil $n$ enak 10, $r$ pa 7.

Vhod v zgoščevalno funkcijo je bil seznam 64 tabel podatkovnega tipa `bytearray`, iz katerih smo izračunali 64 zgoščenih vrednosti. Velikost vsake od tabel je bila od 1 kB do 100 kB.

Na sliki 6.1 je prikazan čas, potreben za izračun zgoščene vrednosti z našim pospeševalnikom na čipu FPGA in z uporabo standardne knjižnice. Pospeševalnik zgoščuje podatke s hitrostjo okoli okoli 200 MB/s, standardna

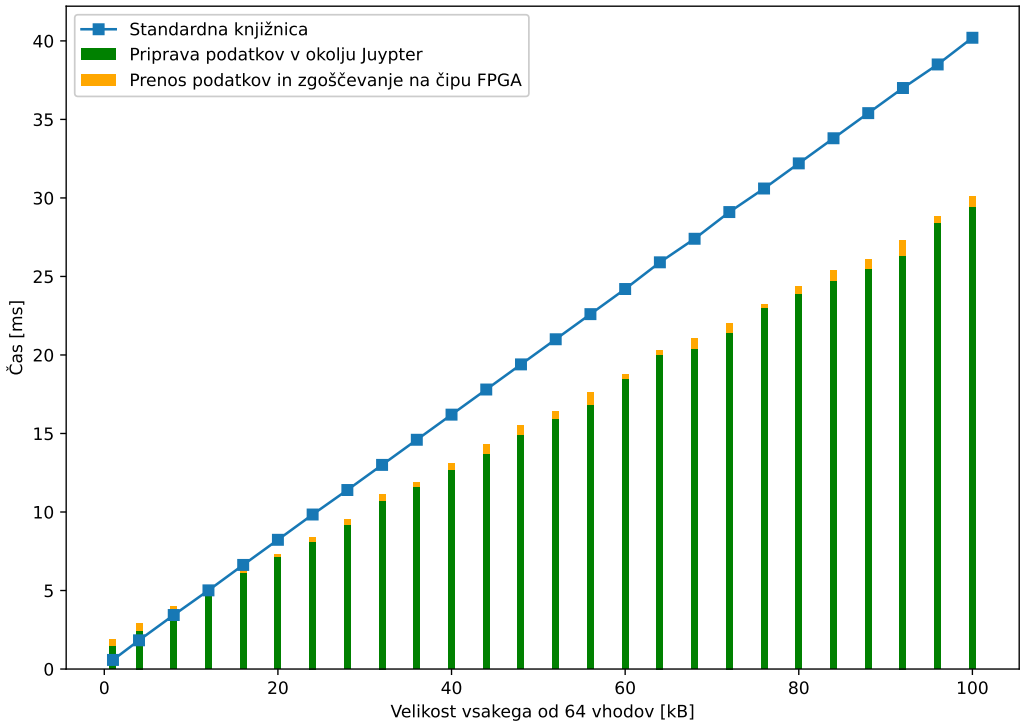
knjižnica pa okoli 160 MB/s. To pomeni, da je pospeševalnik na vezju FPGA okrog 25 % hitrejši od standardne knjižnice.




Slika 6.1: Čas, potreben za izračun zgoščenih vrednosti za 64 vhodov. Napake so reda nekaj mikrosekund, zato jih na grafu nismo prikazali.

Na sliki 6.2 smo delovanje našega algoritma razdelili na čas, ki je potreben za pripravo podatkov v okolju Jupyter ter čas za obdelavo na čipu FPGA (prenos na čip FPGA, zgoščevanje v cevovodu ter prenos nazaj v dinamični pomnilnik). Vidimo lahko, da se velika večina časa izvajanja porabi za pripravo podatkov in ne zgoščevanje. Razlog za to je počasno izvajanje kode, napisane v jeziku Python ter slaba optimizacija predpomnilnika pri kopiranju podatkov v jeziku C.

Največje pridobitve na hitrosti pospeševalnika bi lahko dobili z optimizacijo pomnilniških dostopov pri pripravi podatkov. Počasnemu dostopu do pomnilnika bi se deloma lahko izognili, če bi pripravo podatkov izvajali neposredno na čipu FPGA.




Slika 6.2: Čas, potreben za izračun zgoščenih vrednosti za 64 vhodov, razdeljen na čas priprave podatkov in čas prenosa ter zgoščevanja podatkov. Napake so reda nekaj mikrosekund, zato jih na grafu nismo prikazali.

## 6.1 Omejitve

Naš pospeševalnik je najbolj učinkovit pri zgoščevanju 64 vhodov naenkrat. Pri zgoščevanju manj kot 64 vhodov mora zaradi ohranjanja vrstnega reda blokov v cevovodu število podanih vhodov dopolniti do 64 in tako zgoščevati neuporabne podatke ter njihove zgoščene vrednosti na koncu zavreči. Ohranjanje vrstnega rada blokov v cevovodu je razlog tudi za omejitev, da morajo biti vsi vhodni podatki približno enake dolžine[1]. To omejitev bi lahko odpravili z dodatno logiko, ki bi hitreje izračunane zgoščene vrednosti predhodno zapisala v pomnilnik, vendar zaradi dodatne kompleksnosti rešitve tega nismo implementirali.

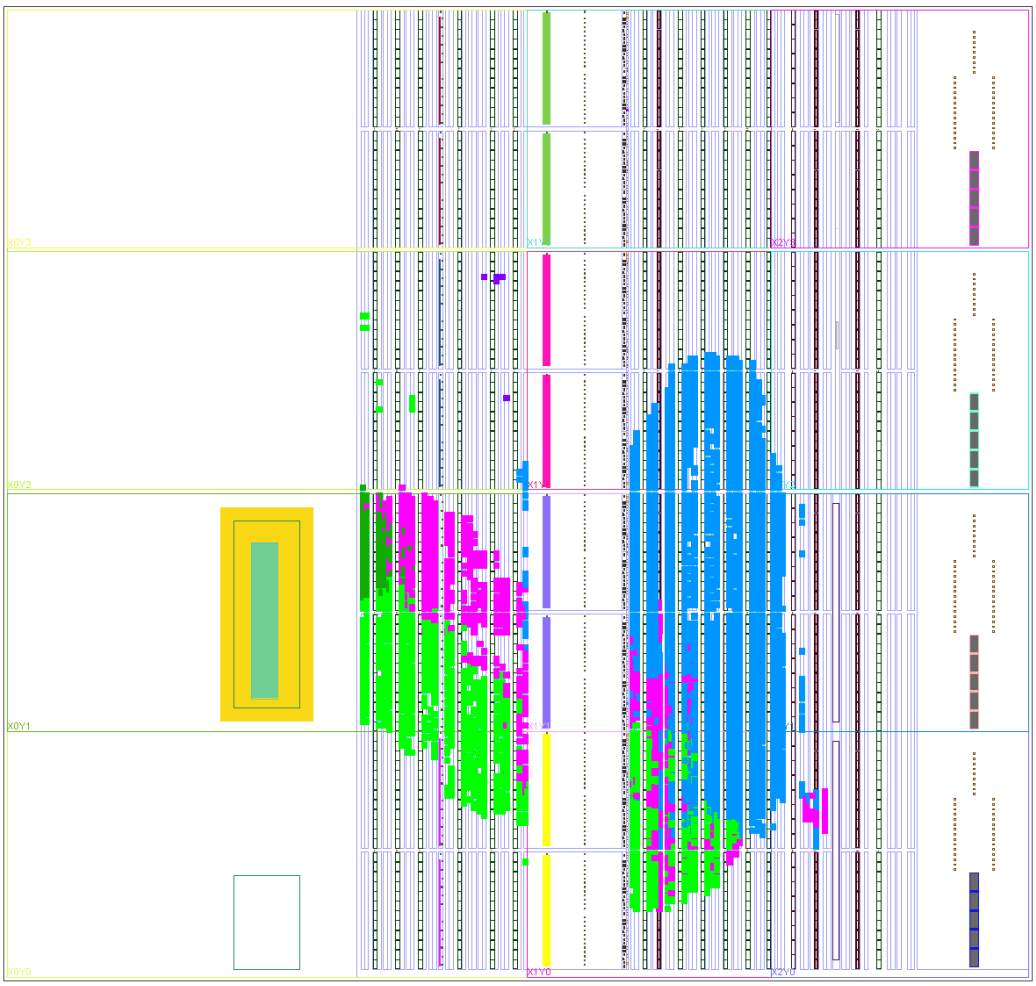
[1]Po koncu dodajanja binarnih ničel se morajo vsi razdeliti na enako število blokov.

Prav tako nismo implementirali zgoščevanja toka podatkov. Pospeševalnik trenutno vse vhodne podatke v celoti prebere v pomnilnik, kar pomeni, da ga ne moremo uporabljati za zgoščevanje vhodov, ki so večji od količine dinamičnega pomnilnika. Zgoščevanje toka podatkov bi delovanje pospeševalnika lahko še pospešilo, saj bi lahko hkrati pripravljali podatke na mikroprocesorju in zgoščevali prej pripravljene na čipu FPGA.

## 6.2 Lastnosti vezja na čipu FPGA

Na sliki 6.3 je prikazan diagram implementiranega vezja na čipu FPGA. Modro je obarvano naše jedro IP za zgoščevanje podatkov z algoritmom MD5. Svetlo in temno zeleno sta obarvani Xilinxovi jedri SmartConnect. Temno zeleno je SmartConnect, ki protokol AXI spremeni v AXI-Lite. Ta mora implementirati le majhno podmnožico vseh funkcionalnosti protokola AXI, zato je veliko manjši kot svetlo zelen SmartConnect, preko katerega potujejo 512-bitni podatki med mikroprocesorjem Zynq in krmilnikom DMA. Krmilnik DMA je obarvan roza, nad njim pa je kot nekaj vijoličnih pik prikazan krmilnik ponastavitvenega signala.

Na levi in desni strani diagrama so prikazane komponente, ki so povezane z vezjem FPGA. Za nas je najpomembnejši mikroprocesor Zynq, ki je prikazan kot zelen pravokotnik z rumeno obrobo na levi strani vezja.



Slika 6.3: Diagram implementiranega vezja na FPGA. Z različnimi barvami so prikazana jedra IP, ki smo jih uporabili pri izdelavi pospeševalnika.

V tabeli 6.1 je zapisana poraba vpoglednih tabel (LUT) in bločnega dinamičnega pomnilnika (BRAM) za vsako uporabljeno jedro IP. Skupno pospeševalnik porabi okoli 16 % kapacitete vpoglednih tabel in 13 % kapacitete bločnega dinamičnega pomnilnika. Največji porabnik je naše jedro za zgoščevanje podatkov.

| **Jedro IP** | **LUT [%]** | **BRAM [%]** |
|---|---|---|
| Zgoščevanje podatkov | 9,63 | 5,21 |
| SmartConnect med mikroprocesorjem Zynq in krmilnikom DMA | 3,44 | 0 |
| Krmilnik DMA | 2,57 | 7,64 |
| SmartConnect za pretvorbo AXI v AXI-Lite | 0,48 | 0 |
| Krmilnik ponastavitvenega signala | 0,01 | 0 |
| **Skupaj** | **16,13** | **12,85** |

Tabela 6.1: Poraba virov na čipu FPGA.

V tabeli 6.2 je prikazana ocenjena poraba električne energije pospeševalnika, ki nam jo je izračunalo razvojno okolje Vivado. Tu je daleč največji porabnik procesor ARM A53, sledi pa mu naše jedro IP za zgoščevanje podatkov. Skupna ocenjena poraba električne energije našega pospeševalnika je okoli 3,7 W.

| **Komponenta** | **Poraba energije [W]** |
|---|---|
| Procesor ARM A53 | 2,7 |
| Jedro IP za zgoščevanje podatkov | 0,42 |
| SmartConnect med mikroprocesorjem Zynq in krmilnikom DMA | 0,16 |
| Krmilnik DMA | 0,02 |
| SmartConnect za pretvorbo AXI v AXI-Lite | <0,01 |
| Krmilnik ponastavitvenega signala | <0,01 |
| Ostalo | 0,39 |
| **Skupaj** | **3,7** |

Tabela 6.2: Poraba električne energije na čipu FPGA.

# Poglavje 7

# Zaključek

V tej diplomski nalogi nas je zanimala težavnost razvoja pospeševalnika z uporabo tehnologije FPGA in če lahko za tak pospeševalnik s knjižnico PYNQ uporabniku ponudimo enako preprost programski vmesnik kot standardna knjižnica.

Na razvojni plošči PYNQ-ZU smo implementirali pospeševalnik za zgoščevanje podatkov z algoritmom MD5. Ta je sestavljen iz prekritja, ki s pomočjo cevovoda zgoščuje podatke z algoritmom MD5 in iz ovojnice za prekritje, ki v okolju Jupyter podatke pripravi na zgoščevanje ter končnemu uporabniku prikaže preprost vmesnik. S tem smo pokazali, da lahko računsko zahtevne algoritme izvajamo na pospeševalniku FPGA, ki je energijsko učinkovitejši kot GPE. Tak pospeševalnik je lahko hitrejši kot standardna knjižnica, pri tem pa ne žrtvuje preprostosti uporabe.

Pri razvoju s knjižnico PYNQ večjih težav nismo imeli, zaradi česar menimo, da ta je primerna za splošno uporabo. Težave smo imeli le pri povezovanju lastnih komponent z že izdelanimi v podjetju Xilinx, saj njihova dokumentacija ni namenjena začetnikom, vendar menimo, da se bo s povečanjem števila razvijalcev v tehnologiji FPGA ta problem sčasoma rešil. Menimo, da sta knjižnica in razvojna plošča PYNQ dober prvi korak v svet tehnologije FPGA, le čas pa bo pokazal če bo njuna uporaba res prešla v splošno uporabo.

# Literatura


[1] *AMBA® AXI-Stream Protocol Specification*. IHI 0051B (ID040921). ARM. 2021.

[2] *AMBA® AXI™ and ACE™ Protocol Specification*. IHI 0022D (ID102711). ARM. 2011.

[3] Shuichi Asano, Tsutomu Maruyama in Yoshiki Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing". V: *2009 International Conference on Field Programmable Logic and Applications*. 2009, str. 126–131. DOI: 10.1109/FPL.2009.5272532.

[4] AWS. *Amazon EC2 F1 Instances*. URL: https://aws.amazon.com/ec2/instance-types/f1/ (pridobljeno 3. 7. 2023).

[5] *AXI DMA v7.1 LogiCORE IP Product Guide*. PG021. Xilinx. Apr. 2022.

[6] Roberto A. Bertolini, Filippo Carloni in Davide Conficconi. "Co-designing an FPGA-Accelerated Encryption Library With PYNQ: The Pynqrypt Case Study". V: *IEEE EUROCON 2023 - 20th International Conference on Smart Technologies*. 2023, str. 683–688. DOI: 10.1109/EUROCON56442.2023.10198938.

[7] Dirk Koch, Frank Hannig in Daniel Ziener. *FPGAs for Software Programmers*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319264060.

[8] Anthony Lancaster. *Beyond Chatbots: The Rise Of Large Language Models*. 20. mar. 2023. URL: https://www.forbes.com/sites/forbestechcouncil/2023/03/20/beyond-chatbots-the-rise-of-large-language-models/?sh=44b1785f2319 (pridobljeno 3. 7. 2023).

[9] Eriko Nurvitadhi in sod. "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC". V: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, str. 1–4. DOI: 10.1109/FPL.2016.7577314.

[10] *PYNQ: Python productivity for Adaptive Computing Platforms*. Ver. 3.0.0. Xilinx. 2022.

[11] Sebastian Raschka, Joshua Patterson in Corey Nolet. "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence". V: *Information* 11.4 (2020). ISSN: 2078-2489. DOI: 10.3390/info11040193. URL: https://www.mdpi.com/2078-2489/11/4/193.

[12] Ronald L. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Apr. 1992. DOI: 10.17487/RFC1321. URL: https://www.rfc-editor.org/info/rfc1321.

[13] Xiaoyun Wang in Hongbo Yu. "How to break MD5 and other hash functions". V: *Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24*. Springer. 2005, str. 19–35.

[14] Benjamin Welte in Joseph Zambreno. "An FPGA Implementation of SipHash". V: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2023, str. 63–70. DOI: 10.1109/IPDPSW59300.2023.00022.

[15] Wikipedia contributors. *MD5 — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/MD5 (pridobljeno 9. 6. 2023).

[16] Xilinx. *Getting Started - Python productivity for Zynq*. URL: http://www.pynq.io/ (pridobljeno 6. 6. 2023).