UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Pustoslemšek

Strojna izvedba zajemanja podatkov za gradnjo histograma v realnem času

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Nikolaj Zimic

Ljubljana, 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01836/2012
Datum: 03.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK PUSTOSLEMŠEK**

Naslov: **STROJNA IZVEDBA ZAJEMANJA PODATKOV ZA GRADNJO HISTOGRAMA V REALNEM ČASU**

HARDWARE IMPLEMENTATION OF DATA ACQUISITION FOR THE CONSTRUCTION OF HISTOGRAM IN REAL TIME

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Obdelava rezultatov meritev v realnem času pri zelo velikih hitrostih zajemanja podatkov je zelo zahtevno opravilo. Ker so hitrosti zajemanja podatkov zelo velike, je potrebno izdelati strojno izvedbo obdelave meritev.

V diplomski nalogi na osnovi XILINX programabilnega vezja izdelajte aplikacijo, ki bo zajemala podatke iz zunanjega okolja in gradila histogram. Zaradi velike hitrosti zajema podatkov, bodite pozorni na hkratni dostop do pomnilnika. Slednji vsebuje informacije o histogramu s strani vezja, ki gradi histogram, ter USB vezja, ki služi za komunikacijo z računalnikom.

Mentor:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Rok Pustoslemšek,

z vpisno številko 63050092,

sem avtor/-ica diplomskega dela z naslovom:

Strojna izvedba zajemanja podatkov za gradnjo histograma v realnem času

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom
  prof. dr. Nikolaja Zimica
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 12.10.2012 Podpis avtorja/-ice:

Zahvala

Na tem mestu se najprej zahvaljujem mentorju prof. dr. Nikolaju Zimicu za usmerjanje in napotke pri pisanju diplomske naloge. Zahvaljujem se tudi Roku Štefaniču iz podjetja CosyLab, ki mi je omogočil praktično izvedbo diplomske naloge in mi pomagal z idejami, ko sem naletel na ovire.

Diplomska naloga je samo zaključno dejanje študija, zato se zahvaljujem vsem, ki so me na tej poti spremljali, še posebej Roku in Mitji – brez vajinih argumentov bi bil moj pogled na svet popolnoma napačen.

Brez vaše podpore, Tjaša, Jana in Tomaž, mi ne bi uspelo. Na koncu pa se najlepše zahvaljujem tudi staršema, ki sta mi študij omogočila.

Sonji in Srečkotu

Kazalo

Seznam uporabljenih kratic in simbolov

FPGA - field programmable gate array
VHDL - VHSIC hardware description language
VHSIC - very high speed integrated circuits
AD - analogno–digitalen
CLB - configurable logic block
CMT - clock management tile
DCM - digital clock management
PLL - phase-locked loop
IOB - input/output block
LUT - lookup table
USB - universal serial bus
FIFO - first in, first out
ROM - read only memory
DDR2 SDRAM - double data rate synchronous dynamic random access memory

Povzetek

Cilj diplomske naloge je načrtovati arhitekturo in implementirati aplikacijo gradnje histograma v realnem času na FPGA vezju Xilinx Virtex 5. Delo obsega predstavitev problema, ki ga rešujemo in njegovo rešitev. Poudarek je na načrtovanju arhitekture in predstavitvi problemov pri implementaciji.

Gradnja histograma v realnem času je problem obdelave ogromne količine podatkov, ki jih pridobivamo v realnem času. Rešitev njihove uporabe je v strojni implementaciji aplikacije za njihovo obdelavo. Možnih arhitekturnih rešitev je več, v delu pa je prikazana ena od možnosti. Aplikacija je implementirana v jeziku VHDL, ob tem pa so rešeni problemi signalov pri prečkanju časovnih domen in problemi uporabe podsistemov, ki so prikazani bodisi kot "bela škatla", bodisi kot "črna škatla".

Diplomsko delo predstavi in analizira rezultate implementirane aplikacije. Grafično prikaže prejete histograme in z njihovo pomočjo sklepa, da je bil projekt uspešno zaključen. Prav tako se ugotovi, da je šum iz okolja potencialen vir nenatančnosti, še posebej zaradi velike natančnosti histograma.

Ključne besede:

FPGA, VHDL, hitri AD pretvornik, histogram, načrtovanje arhitekture, realni čas

Abstract

The goal of this thesis is to design architecture for real-time histogram building application on Xilinx Virtex 5 FPGA and to implement it. Thesis presents the problem we are solving and its solution. The emphasis is on architecture design and problem analisys of the implementation process.

Histogram building in real time is a problem of processing large amount of data, which is acquired in real time. For practical use of this data we need to implement an application for that purpose. There are many possible architectural solutions for that problem and we will concentrate on one of them. In the process of implementation in VHDL programming language we will solve the problem of signal time domain crossing and the problems of using either "white box" or "black box" subsystems.

We will conclude the thesis with a presentation and an analisys of the results. We will plot the recieved data as histograms and on that basis we will conclude that we have successfully completed our project. We will also establish that background noise is a source of problems in electronic circuits, especially due to high resolution of the histogram.

Key words:

FPGA, VHDL, fast AD converter, histogram, architecture design, real time

Poglavje 1

Uvod

FPGA (ang. field programmable gate array) programabilna vezja nam omogočajo, da implementiramo povsem namensko računalniško arhitekturo. To nam omogoča načrtovanje uporabniku prilagojenih, kompleksnih in izjemno hitrih aplikacij. Prednost takšnega razvoja strojne opreme je v ceni razvoja na enoto, ki je dovolj nizka za majhne serije specifičnih izdelkov. Ko razvijamo strojno opremo, moramo v vsakem primeru najprej razviti arhitekturo, toda že izdelava majhne serije izdelkov je izjemno draga, zato lahko v tem primeru izkoristimo FPGA programabilna vezja. Vezja so skoraj tako hitra, kot namenska strojna oprema, izdelava sistema osnovanega okoli njih, pa je cenejša. Prav tako je kasnejše nadgrajevanje enostavno, kar smo pravzaprav naredili.

Tako imamo sedaj na voljo orodje za reševanje tako zahtevnih problemov, kot so problemi manipulacije s podatki. Podatkov je vedno več, potrebe po ustrezni in hitri obdelavi pa postajajo vse pomembnejše, še posebej takrat, kadar je množica podatkov za obdelavo teoretično neskončna. Primere takšnih problemov predstavlja opazovanje sistemov na daljše časovno obdobje, kjer podatki fizikalnih meritev dosežejo več terabajtov. Ti problemi zahtevajo načrtovanje aplikacij za obdelavo podatkov v realnem času. Primer takšnega problema je analiziranje zajetih vzorcev hitrega AD pretvornika, povezanega z FPGA programabilnim vezjem.

V diplomski nalogi bomo predstavili proces razvoja aplikacije, ki smo jo načrtovali. Pričeli bomo tako, da bomo matematično definirali histogram in predstavili okolje, v katerem bo naša aplikacija delovala. Vse to nam bo omogočilo, da si bomo lahko zamislili, kako bomo realizirali aplikacijo gradnje histograma in bomo hkrati razumeli zahtevo po procesiranju v realnem času.

Načrtovanje računalniške arhitekture je postopek, pri katerem vzamemo logične gradnike, ki so nam na voljo, in jih povežemo tako, da dobimo željeno

funkcionalnost. V pričujoči diplomski nalogi bomo s takšnim pristopom načrtovali aplikacijo gradnje histograma na FPGA vezju v realnem času. Najprej bomo opisali, kaj pravzaprav je histogram, potem bomo predstavili okolje, v katerem bo delovala aplikacija, in nato opisali, kakšne so funkcijske zahteve aplikacije.

Arhitekturo naprave bomo začeli načrtovati tako, da bomo najprej preučili FPGA vezje s hitrim AD pretvornikom in pripadajočo izvorno kodo. Na voljo imamo že realizirano takšno FPGA vezje in njegovo izvorno kodo, ki nam omogoča komunikacijo z vmesniki, kot sta na primer USB vmesnik in AD pretvornik. Nato bomo podali idejo o tem, kako bi načrtovali arhitekturo. Pomembno je namreč, da pričnemo s točno določenim ciljem v mislih, zato bomo na začetku izdelali grobi načrt, ki nam bo služil kot podlaga za naslednje korake. V posameznih korakih načrtovanja arhitekture se bomo srečali z večjimi problemi, ki jih bomo morali rešiti, da bo naša aplikacija pravilno in dovolj hitro delovala. Ti problemi nam bodo pravzaprav močno diktirali načrtovano arhitekturo in jih moramo razumeti, da bomo lahko pojasnili, zakaj je arhitektura takšna, kot je. Ko izpolnemo vse te pogoje, je načrtovanje arhitekture končano, saj moramo potem le še ustrezno razporediti gradnike v primerne module.

Arhitekturi sledi implementacija. V poglavju implementacije bomo predstavili primer razreza arhitekture na module, ki smo ga uporabili kasneje v aplikaciji. Čeprav na prvi pogled izgleda, kot da je implementacija trivialna naloga, bomo predstavili vse težave in tako ugotovili, da je programiranje v jeziku VHDL pravzaprav težek in časovno zahteven proces. Nekaj težav je precej splošnih, kot je npr. prehod signala iz ene v drugo časovno domeno, nekaj težav pa je tudi povsem specifičnih našemu razvojnemu okolju. Tem težavam bomo namenili več poudarka, saj je časovna zahtevnost načrtovanja aplikacije odvisna prav od tega.

V zaključnem delu bomo predstavili rezultate, torej zajete histograme, zgrajene v testnem okolju. Ugotovili bomo, ali smo projekt izpeljali uspešno, ali ne. Moramo se zavedati dejstva, da je naš glavni cilj pravilno delujoča aplikacija. Samo pravilno delujoča aplikacija upravičuje izvedbo projekta in določa uspešnost našega dela. Rezultate bomo najprej predstavili grafično, nato pa bomo poskušali razložiti posebnosti, ki smo jih opazili.

Poglavje 2

Histogram

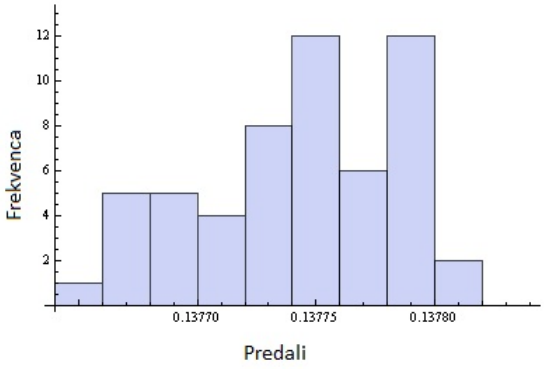
V tem poglavju bomo predstavili matematično definicijo histograma. Po tej predstavitvi bomo pogledali okolje, v katerem bomo gradili naš histogram in ga hkrati natančno preučili. Ker bomo poznali okolje, bomo lahko opisali postopek gradnje histograma v njem. Vse skupaj nam bo omogočilo premislek o tem, kako realizirati aplikacijo za gradnjo histograma v realnem času.

2.1 Definicija in opis histograma

Histogram je v statistiki posebna grafična predstavitev porazdelitve podatkov oziroma dogodkov. Ključna je ena sama spremenljivka, druga pa je navadno definirana nad neko standardno množico npr. dan v mesecu, mesec, relativni čas, zaporedna številka ... V praksi so histogrami prisotni predvsem pri opazovanju dogodkov v nekem časovnem intervalu (npr. gibanje cen, populacije), grafično pa so predstavljeni s stopničastim grafom. Stopničast graf je črta konstantne višine nad intervali spremenljivke na abscisni osi. Te diskretne intervale imenujemo "predali", kot lahko vidimo na primeru histograma na sliki 2.1. Na ordinatni osi prikazujemo število ali frekvenco teh dogodkov, ki pripada določenemu predalu. Širina predalov definira pomembno lastnost histograma, to je njegovo ločljivost. Manjša kot je velikost predala oz. diskretnega intervala, večja je ločljivost histograma. Ločljivost histograma je pomembna glede na posamezen primer; v nekaterih primerih želimo čim večjo ločljivost, v drugih pa je ločljivost manj pomembna in je lahko manjša, da dobimo bolj pregleden graf.

Histogram torej uporabljamo zato, da grafično prikažemo porazdelitev nekih dogodkov preko diskretnih časovnih intervalov. Ta definicija bo za potrebe in razumevanje diplomske naloge povsem zadostna, saj naš cilj predstavlja

gradnja histograma v realnem času.

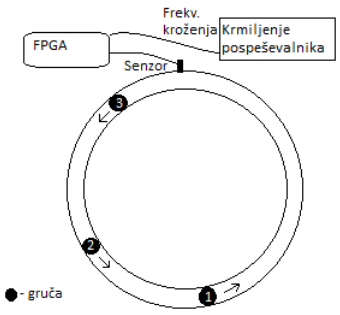


Slika 2.1: Primer histograma

2.2 Vhodne spremenljivke okolja

Da bi se odločili, kako bomo realizirali naš projekt gradnje histograma, moramo najprej razumeti in preučiti okolje, iz katerega bomo zajemali meritve za gradnjo histograma. Predstavimo najprej konkretno okolje, prikazano na sliki 2.2. Pri krožnem pospeševalniku delcev vemo, da v krogu krožijo npr. protoni, ki so združeni v gruče. Potrebno je poudariti, da je lahko v pospeševalniku naenkrat več gruč, ki krožijo z enako hitrostjo. Iz našega okolja dobimo dva signala; prvi signal je sinhroniziran s kroženjem gruč, prehod tega signala iz nizkega v visoko stanje pa nam predstavlja neko časovno točko, na kateri začnemo s ciklom gradnje histograma. Frekvenca tega signala je torej enaka frekvenci kroženja gruč, signal pa bomo poimenovali sinhronizacijski signal. Drugi vhodni signal nam predstavlja analogno meritev senzorja neke fizikalne količine, senzor pa je nameščen na nespremenljivem mestu na pospeševalniku delcev. Dani so nam podatki o frekvenci kroženja gruč in minimalni čas med prehodoma dveh zaporednih gruč.

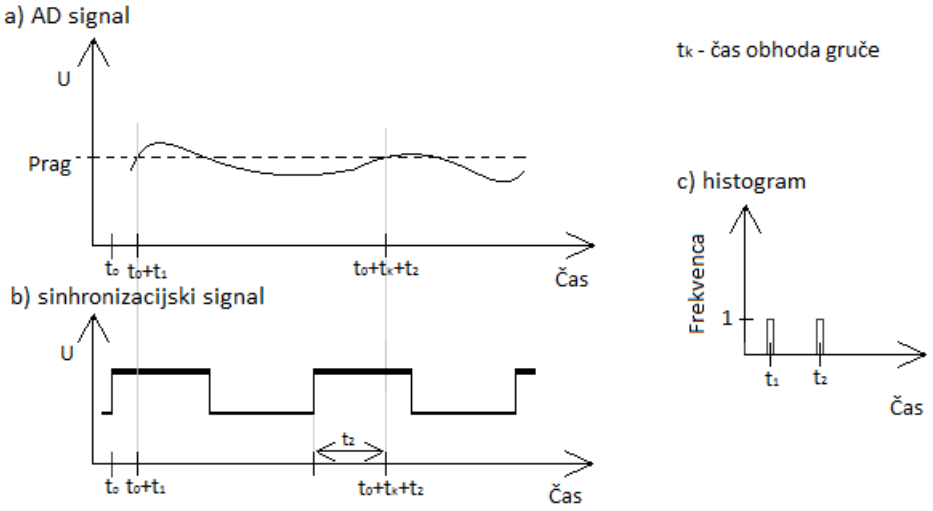
Kot rečeno, je okolje zelo specifično, zato je tudi naša aplikacija temu primerna. Na prvem vhodu imamo diskretni signal, na katerem moramo opazovati pozitivno urino fronto, torej prehod iz nizkega v visoko stanje. Časovni interval med dvema takšnima prehodoma bomo definirali kot cikel histograma. Na drugi vhodni liniji dobimo meritve senzorja, ki meri neko fizikalno količino, odvisno od prehoda gruče, v našem sistemu pa bomo obravnavali vrednosti,



Slika 2.2: Shematski prikaz okolja s tremi gručami v pospeševalniku delcev

ki jih dobimo iz analogno–digitalnega pretvornika. Da bi lahko zgradili histogram, moramo definirati dogodek nad množico meritev senzorja in posledično te dogodke uporabiti za gradnjo histograma. Dogodek bo v našem primeru prehod izhodnega signala senzorja preko neke mejne vrednosti (ang. threshold value). Paziti moramo na nihanje izhodnih vrednosti senzorja okoli mejne vrednosti, hkrati pa moramo biti pozorni na dejstvo, da kot dogodek štejemo samo prehode signala preko mejne vrednosti, ne pa vsake meritve nad mejno vrednostjo. Sedaj lahko z besedami opišemo, kako bomo pravzaprav gradili histogram. Vsakič, ko bomo zaznali pozitivno fronto na sinhronizacijskem signalu, se bomo postavili na začetek histograma. Opazovali bomo, kdaj meritev senzorja v nekem kratkem času prvič preseže določeno mejno vrednost, in ko se bo to zgodilo, bomo v pripadajočem predalu histograma povečali vrednost števca za 1. Ko bo minilo večje število ciklov, bomo lahko videli, ali prihaja do dogodkov ob vsakem prehodu gruče mimo senzorja. To bomo ugotovili tako, da bomo pogledali vrednosti predalov histograma in jih primerjali s številom ciklov. Poleg tega bomo lahko opazovali tudi morebitna časovna odstopanja glede na sinhronizacijski signal.

Slika 2.3 nam prikazuje stanje histograma po dveh ciklih, grafi pa niso dejanske meritve, ampak so namenjeni samo vizualizaciji vhodnih signalov in rešitve – histograma. Graf a) prikazuje primer vhoda napetosti na AD pretvorniku. Glede na nastavljeno mejno vrednost, označeno na grafu kot prag, lahko vidimo, da sta se zgodila dva dogodka. Na grafu b) je prikaz napetosti na vhodu sinhronizacijskega signala, označena pa je tudi točka t_0, ki predstavlja prvo pozitivno fronto. Ob času t_0+t_1 se zgodi prvi dogodek, ob času $t_0+t_k+t_2$ pa drugi dogodek, kjer nam t_k predstavlja čas enega obhoda gruče (1/frekvenca sinhronizacijskega signala). Histogram c) je zgrajen glede na dogodke grafa a) in pozitivno fronto grafa b), saj je izhodišče histograma



Slika 2.3: Primer histograma zgrajenega na opisanem okolju

poravnano s časom t_0.

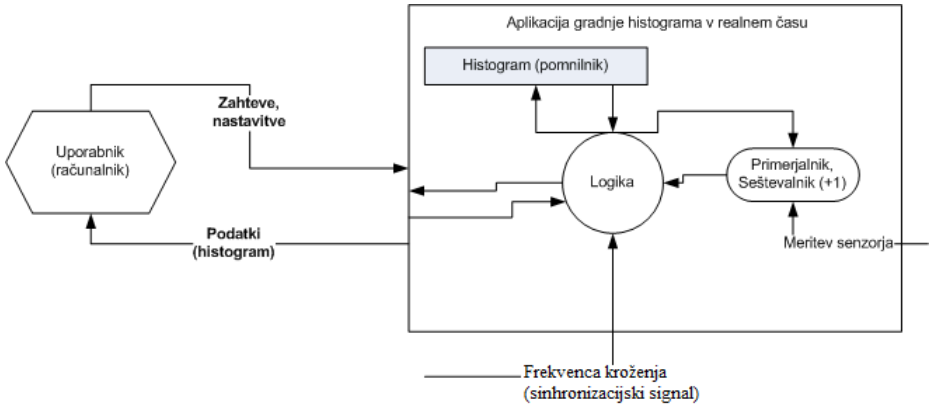
S temi opisi smo definirali pomen vhodnih spremenljivk histograma in opisali tiste pogoje v okolju, ki nas bodo omejevali pri načrtovanju arhitekture aplikacije gradnje histograma v realnem času. Podajmo sedaj še zahtevano in željeno velikost predala histograma. Zahtevamo največ 10 ns širok predal, vendar hkrati načrtujemo ta čas še zmanjšati, kolikor je le-to mogoče. Ker poznamo frekvenco kroženja gruč, lahko izračunamo, da bo imel histogram več 1000 predalov. Več bomo o tem govorili v poglavju o arhitekturi sistema, saj smo te številke samo omenili, da dobi bralec občutek o časovni zahtevnosti problema.

Naredimo sedaj primerjavo navedenih časov s hitrostjo svetlobe. Kot vemo, svetloba v 1 ns potuje približno 30 cm daleč, za električni signal pa je ta vrednost še manjša. Ko trdimo, da zahtevamo ločljivost histograma manj kot 10 ns, se moramo teh primerjav zavedati. Ugotovimo lahko, da je smiselnost takšne natančnosti vprašljiva, vendar kot razvijalci sistema ne postavljamo funkcionalnih zahtev, ampak poskušamo kar najbolje izpolniti zahteve naročnika.

2.3 Gradnja histograma v realnem času

Do te točke že poznamo okolje in vemo, kaj želimo prikazati na našem histogramu. Lahko bi samo izvedli meritve izhoda senzorja s kakšnim osciloskopom, primerno upoštevali pozitivne prehode urine fronte na prvem vhodu, in spisali aplikacijo, ki bi nam na računalniku izrisala histogram s temi zajetimi podatki. Vendar se hitro pojavi omejitev količine zajetih podatkov. Če želimo graditi histogram preko daljšega časovnega obdobja, zajamemo s tem nepredstavljivo količino vzorcev na AD pretvorniku, ki jo je nemogoče shraniti brez uporabe pomnjenja na trdi disk. Rešitev problema takšne ogromne količine podatkov je gradnja histograma v realnem času.

V realnem času bomo primerjali, če vzorec z AD pretvornika sega preko mejne vrednosti, ki jo nastavi uporabnik kot vrednost v registru. V kolikor je vzorec večji od mejne vrednosti, hkrati pa gre za prvi prehod preko le-te v nekem kratkem časovnem intervalu, to obravnavamo kot dogodek in števcu primernega predala histograma povečamo vrednost za 1. Pri tem moramo paziti na časovno poravnanost dogodkov, relativno na pozitiven prehod urine fronte na sinhronizacijskem signalu. Želimo doseči kar najmanjšo časovno velikost predala histograma, kar lahko dosežemo z dobro zasnovano arhitekturo. Arhitektura pa je seveda odvisna od razpoložljivih resursov, lastnosti danega FPGA vezja s hitrim AD pretvornikom, na katerem bomo realizirali gradnjo histograma v realnem času, in željenih dodatnih lastnosti sistema.



Slika 2.4: Grob načrt aplikacije za gradnjo histograma v realnem času

Na sliki 2.4 smo prikazali grobo arhitekturo, ki je posledica zgornjega opisa. Na vsakem nadaljnjem koraku bomo bolj natančno načrtovali arhitekturo in premislili, kakšne rešitve so boljše. Uporabnik bo nastavil napravo in ob na-

ključnih časih zahteval prenos trenutnega histograma na računalnik. Histogram bo shranjen v pomnilniku, kjer bomo vsak predal predstavili kot 32-bitni števec, ob ugotovitvi dogodka s primerjanjem vzorcev z AD pretvornika pa bomo povečali vsebino števca, ki pripada ustreznemu predalu histograma.

Poglavje 3

Programabilna vezja FPGA

Na začetku moramo predstaviti platformo, na kateri bomo realizirali našo aplikacijo gradnje histograma v realnem času. Naša osnova je FPGA vezje s hitrim AD pretvornikom, osnovano okoli čipa Xilinx Virtex 5. Imamo izvorno kodo za komunikacijo z vmesniki FPGA vezja s hitrim AD pretvornikom in v gonilnik vgrajene funkcije, s katerimi bomo lahko upravljali in komunicirali z napravo. Vezja FPGA bomo predstavili zelo specifično, osredotočili se bomo predvsem na lastnosti, ki pomembno vplivajo na načrtovanje naše arhitekture. Namen diplomske naloge je predvsem predstaviti proces načrtovanja arhitekture in izvedbo projekta.

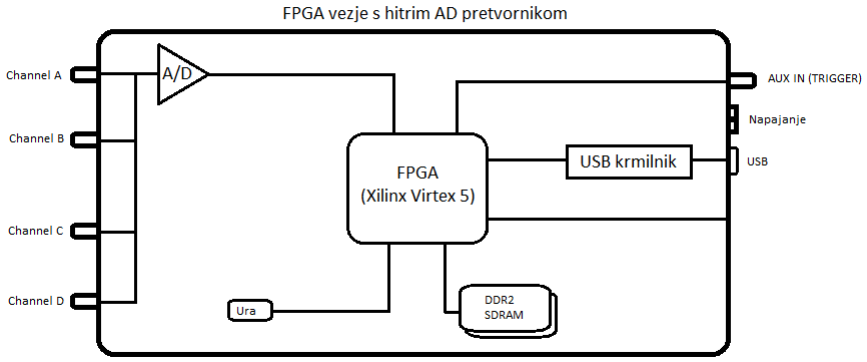
FPGA programabilna vezja so sestavljena iz različnih funkcijskih blokov. Največ je konfigurabilnih logičnih blokov (CLB), ki so namenjeni realizaciji poljubnih logičnih funkcij. Poleg teh blokov imamo še specialne bloke, izmed katerih smo v projektu uporabili bloke za upravljanje z uro (CMT), bloke hitrega pomnilnika velikosti 36 Kb/blok in vhodno-izhodne bloke (IOB). Za pravilno uporabo blokov CLB poskrbi sinteza, ostale bloke pa upravljamo s pomočjo primitivov, ali pa jih implementiramo kot jedra s pomočjo Xilinx IP core-a.

3.1 FPGA vezje s hitrim analogno–digitalnim pretvornikom

Kot že rečeno, FPGA vezje s hitrim AD pretvornikom je zgrajeno okoli čipa Xilinx Virtex 5 XC5VLX50. Na sliki 3.1 lahko hitro ugotovimo, da ima sistem priključke za štiri merilne sonde. Poleg teh vhodov ima še t. i. “aux_in” vhod, ki je za razliko od drugih vhodov digitalen. Ugotovimo lahko, da se na zadnji

strani nahajata USB vmesnik in priključek za napajanje. V našem sistemu bomo uporabili merilne sonde za meritve senzorja in "aux_in" vhod, na katerega bomo priklopili linijo sinhronizacijskega signala. Preko USB vmesnika bomo komunicirali z računalnikom.

Orodje Xilinx nam zgenerira "bitstream" datoteko. To je datoteka, ki nastavi FPGA čip, kot smo ga sprogramirali. Naše FPGA vezje s hitrim AD pretvornikom ima prednaloženo datoteko že v notranjem pomnilniku, sistem pa v tem primeru deluje tako, kot je opisano v priloženi dokumentaciji. Poleg tega imamo na voljo možnost, da nadomestimo to "bitstream" datoteko z našo lastno, torej z aplikacijo gradnje histograma v realnem času. Ker se naša datoteka ne shrani v notranji pomnilnik, moramo ta postopek izpeljati vsakič, ko ponovno zaženemo napravo in želimo uporabljati našo funkcionalnost.



Slika 3.1: Osnovni gradniki FPGA vezja s hitrim AD pretvornikom

V tem delu lahko podamo tudi nekatere ugotovitve, ki smo jih pridobili s pregledom izvorne kode FPGA vezja s hitrim AD pretvornikom. Zanimiva je frekvenca zajemanja vzorcev, saj ugotovimo, da je AD pretvornik sposoben zajemati vzorce s frekvenco 5 GHz. Ta frekvenca vzorčenja je dosežena samo pod pogojem, da je omogočen samo en kanal za zajemanje vzorcev. Če sta omogočena dva kanala, se frekvenca vzorčenja zmanjša na 2,5 GHz, ob omogočenih vseh kanalih pa je ta frekvenca 1,25 GHz. Na sliki 3.1 lahko vidimo osnovne gradnike opisanega sistema, ki so nam na voljo za razvoj naše aplikacije.

3.2 Programabilno vezje Xilinx Virtex 5

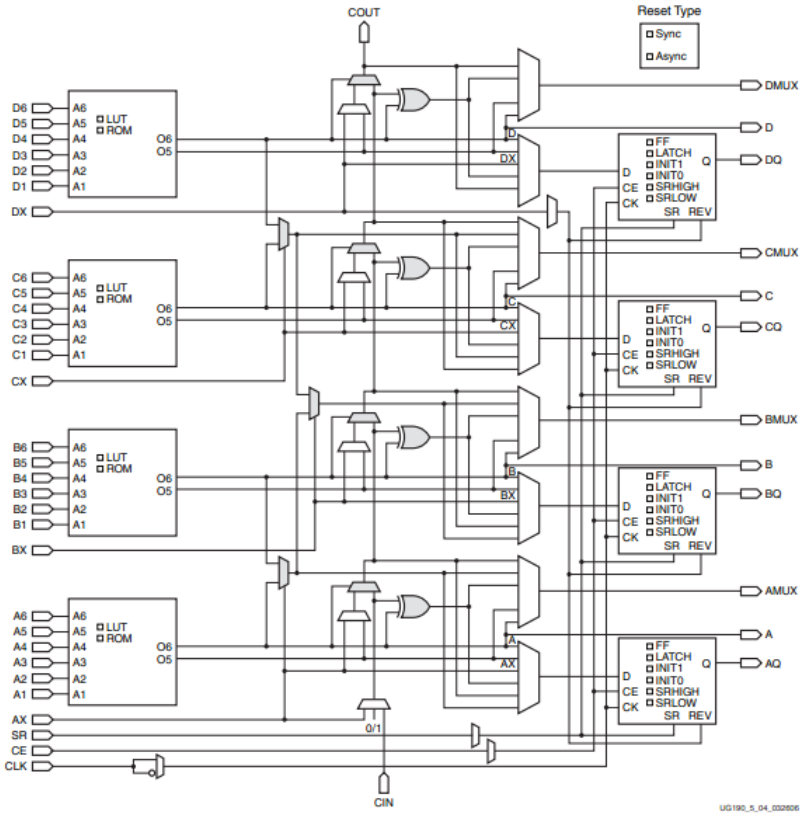
Virtex 5 je FPGA družina čipov podjetja Xilinx, izmed katerih bomo uporabili model XC5VLX50, viden na sliki 3.2. Specifikacije čipa lahko najdemo v literaturi [1], za naš projekt pa je najbolj pomemben podatek o številu elementov, imenovanih "block ram". Gre za hitri statični pomnilnik, ki ga uporabljamo kot naslovljivi hitri pomnilnik, hitri pomnilnik z implementirano FIFO (ang. first in, first out) čakalno vrsto, RAM z dvojnimi vrati in podobno. Zgoraj omenjeni čip ima 3600 blokov CLB, 48 blokov Block RAM in 6 blokov CMT. V nadaljevanju bomo povzeli lastnosti blokov, kot so opisane v literaturi [2].

Slika 3.2: Xilinx Virtex 5 XC5VLX50

3.2.1 Konfigurabilni logični bloki

Kot že omenjeno, konfigurabilni logični bloki vsebujejo gradnike za realizacijo odločitvenih in sekvenčnih operacij. Odločitvene funkcije so realizirane s pomočjo posebnih preslikovalnih tabel, imenovanih LUT (ang. look-up table), v tej družini čipov pa ima vsak CLB dve rezini s po štirimi LUT tabelami. Gradnike rezine lahko vidimo na sliki 3.3. LUT tabela je posebno vezje s šestimi vhodi in dvema izhodoma, s katerimi realiziramo poljubno 6-vhodno Boolean funkcijo. V tem primeru je uporabljen samo en izhod. Če pa želimo implementirati dve 5-vhodni Boolean funkciji, pa sta aktivna oba izhoda LUT tabele. Poleg LUT rezine vsebujejo še tri multiplekserje, kar omogoča realizacijo sedem- in osemvhodnih Boolean funkcij. Vsaki LUT pripada tudi D

flip-flop, ki omogoča pomnjenje. Ker za realizacijo odločitvenih funkcij poskrbi sinteza, nam ni potrebno uporabljati primitivov; te uporabimo samo v primeru, kadar nismo zadovoljni z rezultati sinteze.

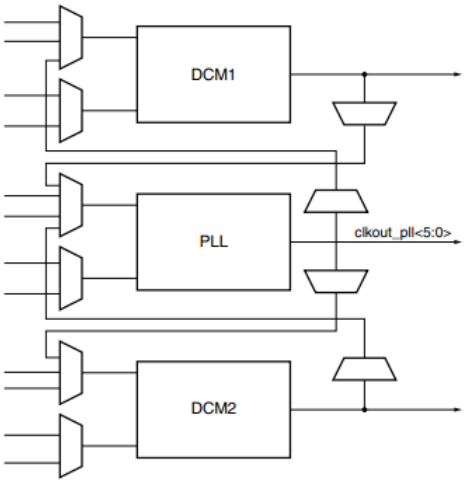


Slika 3.3: Shema rezine konfigurabilnega logičnega bloka [2]

3.2.2 Blok za upravljanje z uro CMT

Bloki CMT (ang. clock management tile) so bloki namenjeni generiranju urinega signala, sinhroniziranega izvornemu urinemu signalu, ki služi kot vhod v bloke CMT. Vsak blok CMT vsebuje dva bloka DCM (ang. digital clock management) in en blok PLL (ang. phase-locked loop). O tem se lahko prepričamo na sliki 3.4. Kot smo že prej zapisali, ima naš model čipa 6 CMT blokov. Te bloke omenjamo zato, ker smo sistem zasnovali tako, da deluje v različnih časovnih domenah. Odločitev za uporabo DCM ali PLL bloka je odvisna od situacije, prednost bloka PLL pa je v večjem in prilagodljivem številu različnih

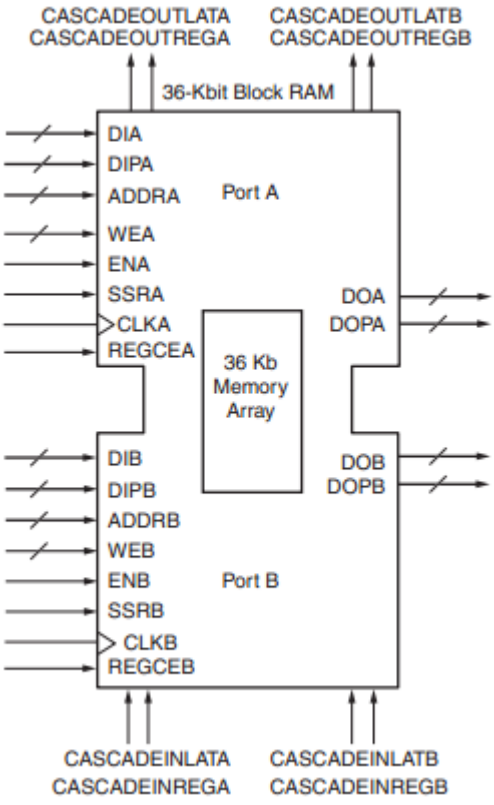
izhodnih signalov ure. Le-teh lahko imamo 6, vsi pa so sinhronizirani z isto referenčno uro. Na drugi strani ima blok DCM vnaprej nastavljene izhodne signale ure glede na referenčno uro in je bolj natančen pri generiranju ure s faznim zamikom.



Slika 3.4: Bločni diagram bloka CMT pri čipih Virtex 5 [2]

3.2.3 Block RAM blok

Xilinx Virtex 5 XC5VLX50 vsebuje 48 block RAM blokov velikosti 36Kb, kar skupno znaša 1728 Kb prostora. Vsak 36 Kb blok vsebuje dva neodvisno kontrolirana 18 Kb RAM-a. Bloki so locirani v stolpec, kar omogoča kaskadno združevanje v širše in globlje bloke RAM-a z zelo malo časovne kazni. S pomočjo Xilinx IP core generatorja jeder lahko enostavno generiramo eno- in dvovratne RAM-e (slika 3.5), ROM-e, pomnilnike z implementirano FIFO čakalno vrsto in pretvornike podatkovne širine (ang. data width converters). Dodatni strojni resursi omogočajo te implementacije brez dodatno porabljenih blokov CLB. Bločni RAM se v praksi uporablja za prenos podatkov iz ene v drugo časovno domeno. Ker bomo tudi v našem sistemu uporabljali različne časovne domene in iz ene v drugo prenašali podatke, lahko to najlažje naredimo s hitrim pomnilnikom, ki implementira FIFO čakalno vrsto, kjer so vrata za pisanje na eni časovni domeni, vrata za branje pa na drugi.



Slika 3.5: Podatkovna shema bločnega RAM-a z dvojnimi vrati [2]

Poglavje 4

Arhitektura

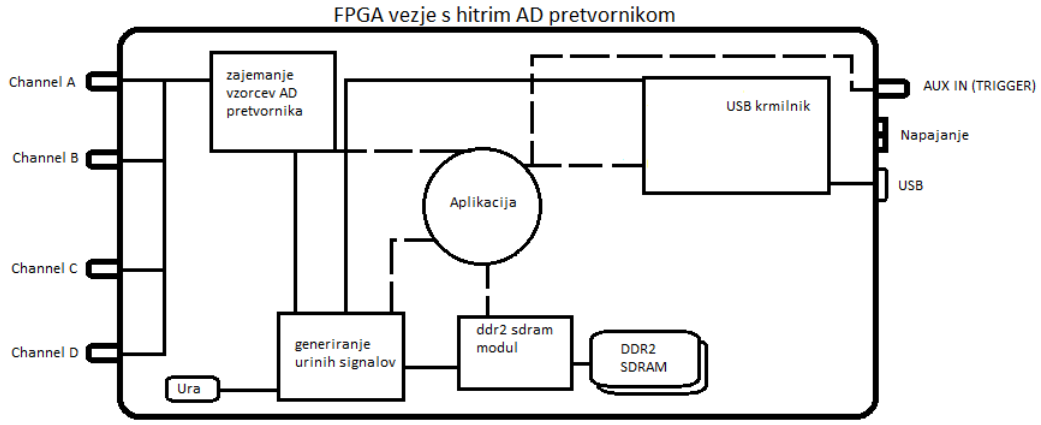
V tem trenutku že poznamo naš problem oz. vemo, kaj moramo narediti in kaj bomo za to uporabili. Tako pridemo do vprašanja, kako bomo to realizirali. Načrtovati moramo torej arhitekturo aplikacije za gradnjo histograma v realnem času. Projekt bomo najprej pričeli tako, da bomo preučili vmesnik FPGA vezja s hitrim AD pretvornikom oziroma izvorno kodo. Zatem bomo najprej grobo načrtovali arhitekturo in nato definirali nastavitve oziroma potrebne registre v registrskem prostoru, da bo aplikacija gradnje histograma v realnem času delovala. Ko bomo zaključili s prej opisanimi koraki, bomo končno izpolnili vse predpogoje za globlje načrtovanje arhitekture. Razmislili bomo o tem, kje so omejitve aplikacije, in arhitekturo načrtovali temu primerno.

Potrebno je razumeti, da je na koncu poglavja predstavljena arhitektura rezultat postopnega razvijanja in popravljanja začetne ideje. V praksi smo arhitekturo razvili, jo poizkusili realizirati, popravili napake in ta postopek ponavljali, dokler nismo bili zadovoljni z izdelkom. V diplomski nalogi bomo predstavili samo trenutno, zadnjo verzijo.

4.1 Pregled izvorne kode FPGA vezja s hitrim AD pretvornikom

Preučitev in razumevanje izvorne kode FPGA vezja s hitrim AD pretvornikom je temelj za vse nadaljnje korake, saj je to delo nekega drugega programerja. Ob pregledu izvorne kode lahko opazimo, kako so generirani različni urini signali, kako poteka prenos podatkov preko USB vmesnika, kako poteka zajemanje vzorcev AD pretvornika in kako je nastavljen modul za dostop do dinamičnega pomnilnika. Ta modul sicer zgenerira orodje Xilinx IP core, ven-

dar pa so nekatere nastavitve specifične glede na uporabljen fizičen pomnilnik. Najprej poglejmo sliko 4.1, kjer lahko vidimo prikaz arhitekture FPGA vezja s hitrim AD pretvornikom, ki jo bomo opisali v nadaljevanju. Neprekinjene črte prikazujejo vgrajeno povezanost med funkcijskimi bloki in vhodi/izhodi naprave, prekinjena črta pa prikazuje, da bomo izhode teh blokov uporabili v naši aplikaciji.



Slika 4.1: Razširjena shema arhitekture FPGA vezja s hitrim AD pretvornikom

Funkcionalnost generiranja urinih signalov je namenjena generiranju različnih urinih signalov, ki jih potrebujemo. S preučitvijo kode lahko vidimo, da naprava za svoje delovanje pravzaprav uporablja uro s frekvenco 48 MHz. Prav tako je generirana ura s frekvenco 156,25 MHz – to je frekvenca, s katero pridobivamo vzorce AD pretvornika. V kolikor se bomo odločili za uporabo DDR2 SDRAM, bomo tukaj lahko generirali urin signal s primerno frekvenco. V FPGA vezjih se za generiranje ure uporabljajo bloki CMT, ki nam generirajo še pripadajoče "reset" signale.

Za potrebe komunikacije z računalnikom preko USB vmesnika je enostavno uporabiti namenski krmilnik, ki ga ima naše FPGA vezje s hitrim AD pretvornikom že vgrajenega. Pri periferni strojni opremi način komunikacije poteka tako, da je zagotovljen nek registrski prostor in omogočen neke vrste bločni prenos podatkov. Ravno takšen način komunikacije z računalnikom nam je na voljo tudi tukaj. S pregledom kode ugotovimo, kako obravnavamo takšen registrski prostor in kako dostavljamo podatke za prenos v večjih kosih. Poleg tega je glavna ugotovitev to, da moramo biti pazljivi pri uporabi kontrolnih signalov pri prenosu podatkov. Boljše razumevanje delovanja USB krmilnika

na tem mestu ni potrebno.

Modul "ddr2 sdram" je glavni modul krmilnika DDR2 dinamičnega pomnilnika, celoten sistem modulov pa lahko tudi zgeneriramo s pomočjo Xilinx IP core. Za razumevanje delovanja krmilnika je potrebno preučiti dodatno literaturo[3], pomembno pa je, da razumemo, kako poteka prenos podatkov. Celoten vmesnik za komunikacijo s pomnilnikom zajema inicializacijski signal, naslovno vodilo, vhodno in izhodno podatkovno vodilo, uro, "reset", ukazne signale in statusne signale. Vidimo lahko, da je podatkovno vodilo širine 256 bitov, zato toliko bitov tudi naslovimo z enim naslovom. Poleg teh signalov lahko nastavimo tudi različne splošne parametre kot je frekvenca osveževanja, velikost prenosa podatkov ob eni zahtevi (ang. burst length) in podobno. Grafi signalov pri branju in pisanju ter razlaga parametrov so na voljo v dokumentaciji[3].

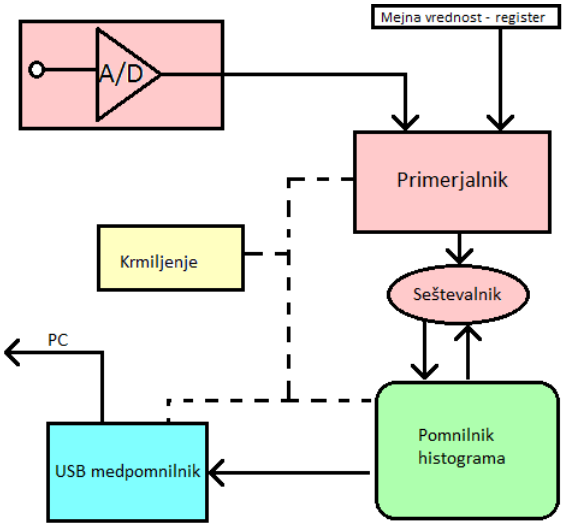
Glede na prej omenjene hitrosti vzorčenja sklepamo, da uporabljamo zunanji hitri AD pretvornik. Po pregledu izvorne kode lahko to potrdimo in ugotovimo s kakšno frekvenco deluje zajemanje. Analogno–digitalni pretvorniki navadno omogočajo zajemanje vzorcev širine 8 bitov in tudi v tem primeru je tako. S pregledom kode razumemo, kako pridobimo zajete vzorce in preučimo njihovo organizacijo na vodilu. Vmesnik nam omogoča, da s frekvenco 156,25 MHz vsako urino periodo dobimo 32 vzorcev AD pretvornika. Kako naprava pravilno vzorči, nas ne zanima, zanima nas samo organizacija vzorcev na podatkovnem vodilu in s kakšno frekvenco se le-ti pojavljajo.

Na tem mestu izvorno kodo FPGA vezja s hitrim AD pretvornikom razumemo oziroma smo seznanjeni z njeno funkcionalnostjo.

4.2 Grob načrt arhitekture

Z izrazom "grob načrt arhitekture" imamo v mislih prvi poskus načrtovanja arhitekture, ne da bi premislili vse podrobnosti. Takšen načrt je potreben, da sploh vemo, kakšna je naša aplikacija in da si pridobimo podlago za bolj temeljito načrtovanje. Ker poznamo lastnosti FPGA vezja s hitrim AD pretvornikom, že lahko predvidimo, kakšen bo tok podatkov in kje nas čakajo problemi.

Našo aplikacijo lahko pogledamo z vidika toka podatkov. Histogram bo shranjen v pomnilniku naprave, bodisi v hitrem bodisi v dinamičnem pomnilniku. Najprej se bo moralo zgoditi branje števca histograma. Vzporedno s prejetim podatkom nam mora biti na voljo tudi rezultat primerjanja nekega števila vzorcev AD pretvornika z željeno mejno vrednostjo. V naslednji urini



Slika 4.2: Grob načrt arhitekture

periodi moramo opraviti popravljanje vrednosti števca histograma in nato spremenjeno vrednost zapisati nazaj v pomnilnik. Poleg tega neprekinjenega toka podatkov moramo tudi periodično posredovati histogram uporabniku. Ob zaključku cikla histograma, torej ob pozitivni fronti sinhronizacijskega signala, pa moramo nadaljevati z gradnjo histograma od začetka, torej od prvega števca naprej.

Če na našo aplikacijo pogledamo z vidika modularnosti, lahko ugotovimo, da potrebujemo več funkcijsko zaključenih modulov. En takšen modul je modul za primerjanje vzorcev AD pretvornika z neko mejno vrednostjo. Ta modul mora vsako urino periodo določiti nov rezultat primerjanja, implementirati moramo torej neke vrste histerezo. Naslednji modul je modul, ki opravlja podatkovni dostop do histograma. V najbolj enostavnem primeru bomo potrebovali samo naslovni števec, če pa ne bo mogoče opraviti enega dostopa vsako urino periodo, pa se bo kompleksnost modula močno povečala. Funkcionalnost pošiljanja histograma uporabniku lahko implementiramo v lastnem modulu, saj je pomembno, da to opravilo izvajamo, medtem ko se histogram nemoteno gradi naprej. Prav tako bomo potrebovali modul, ki bo skrbel za krmiljenje vseh modulov in tako omogočil pravilno delovanje.

Krmiljenje naprave bomo izvedli preko že poznanega registrskega niza, s katerim bomo sprožili začetek gradnje histograma in prožili pošiljanje k uporabniku. Določili bomo še druge registre, s katerimi bomo nadzorovali delovanje naprave. S temi ugotovitvami smo zaključili z grobim načrtom arhitekture,

ki je prikazan na sliki 4.2.

Sedaj lahko začasno preskočimo na sliko 4.9 in si pogledamo končno arhitekturo našega sistema. Odločili smo se, da bomo histogram pomnilnili v dinamičnem pomnilniku. Ker imamo dovolj prostora, lahko gradimo sočasno dva histograma – prvi se prenaša k uporabniku, drugi pa se gradi. Zaradi tipa pomnilnika potrebujemo več medpomnilnikov, ograniziranih kot FIFO čakalna vrsta, v katere končni avtomat za dostop do pomnilnika zapisuje in iz njih bere podatke. Krmiljenje poskrbi za skladnost toka podatkov, saj moramo še posebej paziti, da se rezultat primerjanja vzorcev AD pretvornika z mejno vrednostjo upošteva ob primernem števcu histograma. Rezultate primerjanja zapišemo v ADC medpomnilnik, ko pa so na voljo vsi potrebni podatki, pa opravimo seštevanje števcev histograma z rezultati primerjanja in skupen rezultat zapišemo v medpomnilnik. Kako smo prišli do takšne arhitekture in zakaj je tako načrtovana, pa bomo pojasnili v naslednjih podpoglavjih.

4.3 Registrski prostor – nastavitve uporabnika

V tem podpoglavju bomo definirali nastavitve, ki jih mora nastaviti uporabnik, da začne sistem z gradnjo histograma. V ta namen nam je na voljo registrski niz. Ta registrski niz lahko poljubno uporabljamo kot statusne in kontrolne registre. Poleg nastavitev preko kontrolnih registrov lahko beremo tudi stanje naprave preko statusnih registrov. To je pomembno, da ugotovimo, ali naprava trenutno gradi histogram, ali poteka prenos histograma z naprave k uporabniku in podobno. Registre izbiramo poljubno, edini kriterij je, da z njimi zagotovimo krmiljenje sistema. Za našo aplikacijo so nekateri registri podani v tabeli 4.1.

Najprej potrebujemo "global start" register, v katerem nam pisanje vrednosti 0 ustavi aplikacijo, pisanje vrednosti 1 pa prične z gradnjo histograma. Poleg tega registra potrebujemo še štiri registre, v katere bomo zapisali mejno vrednost (ang. threshold value). Štiri registre potrebujemo zato, ker ima FPGA vezje s hitrim AD pretvornikom štiri kanale. Potrebujemo tudi kontrolni register, s katerim krmilimo prenos histograma k uporabniku. Poleg teh registrov je zmeraj smiselno imeti še dodatne registre, ki so namenjeni kakšni dodatni razširitvi aplikacije.

S pomočjo statusnih registrov uporabnik ugotavlja, kaj se pravzaprav dogaja z napravo. Izmed statusnih registrov potrebujemo register, ki nam bo podal stanje naprave, torej če deluje ali ne. Poleg tega moramo imeti možnost brati zaporedno številko cikla histograma, ki ga želimo prenesti. Prav tako potrebujemo register, ki nam bo povedal, ali trenutno poteka kakšen prenos

ime registra
kontrolni registri
"global start"
mejna vrednost
število ciklov histograma
zahteva prenosa histograma
druge nastavitve
statusni registri
napake notranjih medpomnilnikov
zaporedna številka cikla
velikost histograma
stanje prenosa histograma

Tabela 4.1: Tabela izbranih registrov in naslovov

histograma od naprave k uporabniku. Zaželjen je tudi register, kjer lahko preberemo ali je prišlo do kakšne napake pri uporabi blokov notranjega pomnilnika (ang. underflow/overflow). Nujno potreben je tudi register o številu podatkov za prenos, ko uporabnik zahteva prenos histograma na računalnik. Mi namreč ne samo da ne vemo, točno koliko predalov predstavlja en histogram, ampak želimo narediti sistem bolj splošen in neodvisen od frekvence sinhronizacijskega signala. Frekvenca tega signala in časovna širina histograma nam namreč določata velikost histograma oz. količino podatkov za prenos.

4.4 Omejitve pri načrtovanju arhitekture

V naslednjih podpodpoglavjih bomo ugotovili, katere so tiste omejitve pri načrtovanju arhitekture, na katere moramo biti posebej pozorni.

4.4.1 Pomnjenje histograma

Na začetku načrtovanja arhitekture moramo najprej rešiti vprašanje pomnjenja histograma. Na voljo imamo bločni ram in dinamični pomnilnik DDR2 SDRAM. Omejitev blok rama je v njegovi velikosti, SDRAM-a pa v njegovi hitrosti. Kako se naj torej pravilno odločimo, da bomo dosegli čim manjšo časovno velikost predala histograma? Z zmanjševanjem časovnega intervala predala namreč pridobivamo na številu števcev, kar pomeni več podatkov,

sama odločitev za uporabo DDR2 SDRAM-a pa nam močno oteži načrtovanje arhitekture, saj dostop do podatkov ni več tako enostaven.

Pri blok ramu si bomo pomagali s preprostim izračunom; glede na velikost blok rama bomo izračunali najmanjšo možno velikost predala. Vemo, da imamo največ 1728 Kb prostora, vendar smo ga že nekaj porabili za realizacijo medpomnilnika, namenjenega prenosu podatkov v smeri proti računalniku, a to dejstvo bomo zanemarili. Če ta prostor razdelimo na 32-bitne števce, ugotovimo, da imamo prostora za 55296 le-teh. Na tem mestu bi potrebovali podatek o frekvenci sinhronizacijskega signala, vendar bomo ta podatek obravnavali kot spremenljivko. S pomočjo podatka o časovni širini predala in frekvence sinhronizacijskega signala lahko izračunamo, koliko števcev potrebujemo. Večja kot je frekvenca in manjša kot je časovna širina predala, večji je histogram. Pomembno je tudi spoznanje, ali bomo gradili sočasno histograme za vsak kanal AD pretvornika, ali ne. Na tem mestu lahko ugotovimo, da nam uporaba hitrega pomnilnika res odstrani velik kos kompleksnosti pri načrtovanju, vendar pa to plačamo s pomembno omejitvijo pri uporabi. Ker imamo na voljo relativno malo prostora, moramo postaviti zgornjo mejo maksimalne frekvence sinhronizacijskega signala, ki je precej nižja, kot pri DDR2 SDRAM-u. Tej omejitvi se želimo na vsak način izogniti. Tudi izračun s frekvenco določenega pospeševalnika delcev je pokazal, da je sicer časovna širina predala histograma manjša od 10 ns, vendar večja, kot jo lahko pričakujemo ob uporabi dinamičnega pomnilnika.

Pri DDR2 SDRAM-u nimamo problemov z razpoložljivim prostorom, ampak z maksimalnim pretokom podatkov, kar lahko izračunamo samo z dolgotrajno, poglobljeno analizo. Upoštevati moramo več faktorjev, in sicer frekvenco ure podatkovnega vodila, zaporednost podatkov, velikost prenešenih podatkov na en ukaz, čas osveževanja ... Zaradi toliko faktorjev bomo poskušali samo s približnim izračunom in sklepanjem. Najprej lahko ugotovimo frekvenco delovanja krmilnika dinamičnega pomnilnika in širino podatkovnega vodila v našem FPGA vezju s hitrim AD pretvornikom. Iz dokumentacije Xilinx IP core sledi, da je maksimalna frekvenca takšnega krmilnika 333 Mhz [3]. Ker dostopamo do krmilnika z dodatno logiko v obliki avtomata, smo mnenja, da takšne frekvence ne bomo dosegli, vendar upravičeno pričakujemo frekvenco, ki bo zagotovo večja od ure frekvence vmesnika AD pretvornika. Prav tako v dokumentaciji najdemo podatek, da je največja možna širina podatkovnega vodila dinamičnega pomnilnika 256 bitov [3].

Kot že omenjeno, iz dokumentacije [3] sledi, da je širina podatkovnega vodila 256 bitov. Če te bite razdelimo na 32-bitne števce, lahko ugotovimo, da v eni urini periodi beremo ali pišemo osem števcev. Če ob tem izhajamo iz

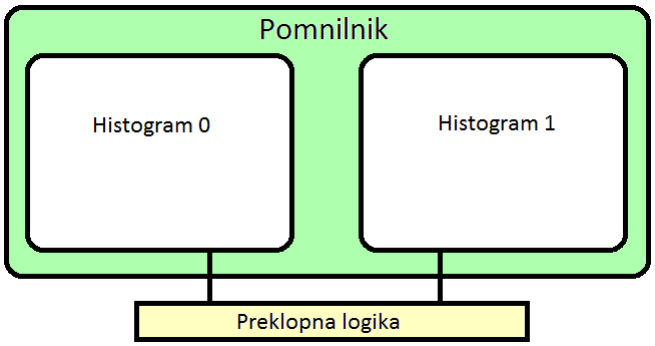
enake predpostavke kot pri blok ramu, tj. da vzporedno gradimo štiri histograme, potem to pomeni dva števca v eni urini periodi na en histogram. V kolikor upoštevamo, da moramo vsak števec posebej prebrati in ga kasneje zapisati nazaj, to pomeni, da bi pri frekvenci krmilnika dinamičnega pomnilnika enaki frekvenci prejemanja novih vzorcev (156,25 MHz) dosegli časovno širino predala 6,4 ns. Za ta izračun ne potrebujemo podatka o frekvenci sinhronizacijskega signala, kar pomeni, da je takšna rešitev bolj splošna in zato tudi boljša.

Pri zgornjem opisovanju nismo upoštevali dodatnega časa, potrebnega za pošiljanje histograma k uporabniku, časa, ki ga porabi krmilnik za osveževanje pomnilnika in časa, ki mine od podane zahteve za podatke krmilniku dinamičnega pomnilnika do takrat, ko se ti podatki pojavijo na podatkovnem vodilu. Naše predvidevanje je, da bomo te časovne zahteve lahko dosegli z zviševanjem frekvence krmilnika dinamičnega pomnilnika. To razmišljanje je podlaga za odločitev, da bomo naredili boljši sistem, če bomo hranili podatke v dinamičnem pomnilniku. To pa močno oteži načrtovanje arhitekture, saj moramo sedaj paziti še na prečkanje časovnih domen.

4.4.2 Konsistenca histograma glede na cikel

Ne glede na odločitev o hranjenju histograma moramo rešiti problem prikazovanja histograma uporabniku. Če želimo uporabniku pošiljati histogram, medtem ko ga aplikacija še naprej gradi, moramo to storiti zelo hitro. Če histograma ne prenesemo hitreje kot je zakasnitev med dvema pozitivnima frontama sinhronizacijskega signala, tj. znotraj enega cikla histograma, potem bo uporabnik dobil histogram, ki je zgrajen v več različnih ciklih. Če npr. histogram prenašamo tako dolgo, kot je zakasnitev med tremi pozitivnimi frontami, potem uporabnik vidi prvo polovico histograma pripadajočo enemu ciklu in drugo polovico pripadajočo naslednjemu ciklu. Tej situaciji se želimo izogniti, če je to le mogoče.

Naša ideja rešitve problema je možna zaradi rešitve pomnjenja histograma v dinamičnem pomnilniku in je prikazana na sliki 4.3. Ker imamo na voljo dovolj prostora, se lahko odločimo, da pravzaprav lahko gradimo dva histograma. Ko uporabnik zahteva prenos histograma, samo nadaljujemo z gradnjo drugega histograma na prostem delu pomnilnika, prvi histogram pa prenašamo k uporabniku. Ob naslednji zahtevi za prenos pa spet nadaljujemo z gradnjo prvega histograma in prenašamo drugega. Tako imamo čas prenašati histogram preko več ciklov, a smo s tem povzročili dodatno delo programerju grafičnega uporabniškega vmesnika. Ko bo uporabnik drugič in vsakič naslednjič zahteval



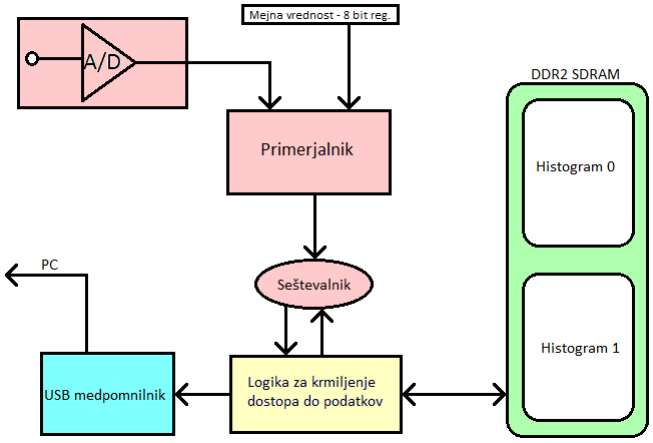
Slika 4.3: Ideja izmenične gradnje dveh histogramov

prenos histograma, bo moral programer za pravilen prikaz sešteti zadnja dva prejeta histograma. Ker želimo konsistenco histograma glede na cikel, sprejmemo takšen kompromis in naložimo dodatno delo programerju grafičnega uporabniškega vmesnika.

4.5 Načrtovana arhitektura

Končno smo definirali vse zahteve in sedaj lahko pogledamo obris popravljene arhitekture na sliki 4.4. Vidimo lahko, kako bo potekal tok podatkov v naši aplikaciji. Najprej zajete vzorce primerjamo, če so večji od vrednosti, zapisane v registru. Izhod tega primerjanja je bodisi 0 bodisi 1, potem pa to vrednost prištejemo primernemu števcu histograma. Da prištejemo pravi rezultat primerjanja k pravemu števcu, mora poskrbeti logika za krmiljenje dostopa do dinamičnega pomnilnika. To je sedaj naša izhodiščna arhitektura, ki jo moramo nadgraditi. Najprej moramo natančno definirati logiko za krmiljenje dostopa do podatkov, ker je potrebno čakanje na podatke in ta logika najbolj poveča kompleksnost sistema. Žal se temu problemu ne moremo izogniti na lep način, ampak moramo uvesti bloke hitrega pomnilnika, ki implementira FIFO čakalno vrsto.

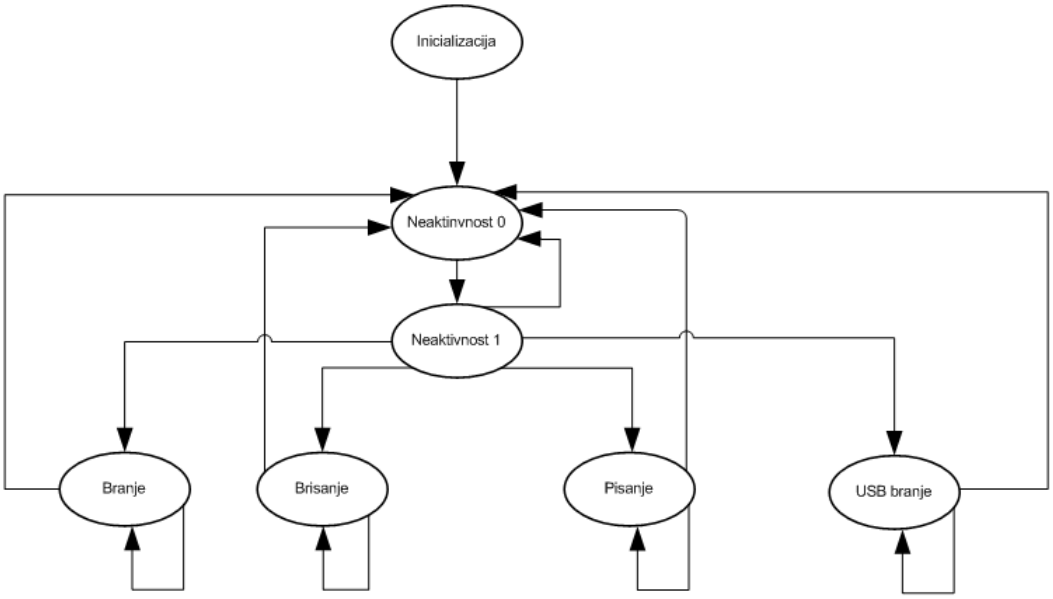
Najprej se odločimo, kako bomo dostopali do dinamičnega pomnilnika. Odločimo se, da bomo realizirali ogromen avtomat s končnim številom stanj (ang. finite state machine). Avtomat je odgovoren za bralno-pisalne dostope do dinamičnega pomnilnika. Zgradimo ga tako, kot je prikazano na sliki 4.5. Vidimo, da avtomat najprej počaka na inicializacijski signal s krmilnika. Zatem čaka v neaktivnem stanju, kjer se odloči, katera operacija bo izvedena naslednja. Najvišjo prioriteto ima brisanje dinamičnega pomnilnika, ki ga sproži



Slika 4.4: Prvi korak načrtovanja arhitekture

uporabnik sam in se izvaja takrat, ko aplikacija ne gradi histograma. Od tega ima nižjo prioriteto branje trenutnega histograma, še manjšo prioriteto ima pisanje trenutnega histograma, najmanjšo prioriteto pa branje histograma za pošiljanje k uporabniku. Takšen vrstni red je potreben zato, da histogram pošiljamo uporabniku med prostim časom in s tem ne motimo gradnje histograma. Ko vstopi avtomat v nek funkcijski del, ostane v njem nekaj ciklov, da minimaliziramo časovne zamude zaradi odpiranja strani DDR2 SDRAM-a. Hkrati je pomembno, da ne izgubljamo podatkov zaradi napolnjenih medpomnilnikov, kar lahko zagotovimo samo s prej omenjenim vrstnim redom prioritet dostopov do dinamičnega pomnilnika. Ker ne obdelujemo podatke histograma z enako frekvenco, kot jih beremo ali pišemo v pomnilnik, uporabimo hitri pomnilnik, ki implementira FIFO čakalno vrsto. Potrebujemo torej vsaj tri bloke medpomnilnika, dva za shranjevanje podatkov, enega pa za branje. Vsi imajo podatkovno vodilo enake širine kot krmilnik dinamičnega pomnilnika, tj. 256 bitov.

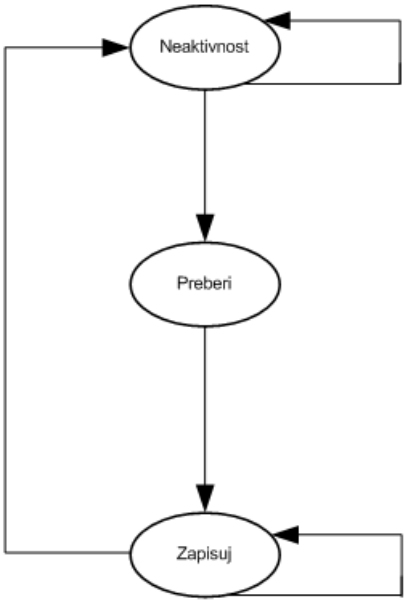
Sedaj se lahko lotimo načrtovanja arhitekture glede na tok podatkov. Najprej moramo ugotoviti, kako bomo zapisali 256 bitov široke podatke, ki so namenjeni za prenos k uporabniku, v USB medpomnilnik podatkovne širine 8 bitov. Kot smo prej omenili, lahko bločni RAM uporabimo tudi za pretvarjanje podatkovne širine, vendar pa nam Xilinx IP core omogoča samo pretvarjanje iz 256 v najmanj 32 bitov. Če želimo še teh 32 bitov pretvoriti v 4-krat po 8 bitov, moramo uporabiti še en bločni RAM, vendar pa smo se odločili, da bomo raje načrtovali še en avtomat, prikazan na sliki 4.6. Ta avtomat bo prebral 256 bitov iz “transfer” medpomnilnika, v katerega piše podatke prejšnji



Slika 4.5: Stanja avtomata za dostop do pomnilnika

avtomat, in jih bo zapisal v kosih po 8 bitov v USB medpomnilnik, namenjen prenosu k uporabniku. Pogoja za to sta, da je vsaj en zapis na voljo v prvem medpomnilniku in 32 prostih mest na voljo v USB medpomnilniku.

Kako pa bomo poskrbeli za preklopno logiko, s katero bomo izmenično gradili histograma? Števci histograma bodo v pomnilniku organizirani tako, da bodo shranjeni na zaporednih lokacijah, tako kot si sledijo v histogramu. Za naslavljanje teh števcev potrebujemo naslovne števce, ki bodo ob vsakem dostopu do pomnilnika povečevali naslov, preklapljanje med histogramoma pa lahko izvedemo preprosto z invertiranjem zgornjega bita tega naslovnega števca. Potrebujemo štiri števce, tako kot imamo tudi štiri glavne veje v avtomatu dostopa do pomnilnika. Števec za brisanje se bo samo zavrtel preko vseh lokacij pomnilnika in takrat bomo na vse lokacije zapisali 0. Števca branja in pisanja imata zgornji naslovni bit enak, saj beremo in pišemo isti histogram, števec prenosa histograma k uporabniku pa ima zgornji naslovni bit invertiran. Ob zahtevi uporabnika za prenos histograma pa poskrbimo, da se zgornji naslovni biti v teh treh števcih pravočasno invertirajo. Na sliki 4.7 smo prikazali do sedaj načrtovano arhitekturo. Sicer še nismo rešili vseh problemov, predvsem kaj se zgodi ob koncu cikla histograma, vendar bomo to rešili kasneje z raznimi kontrolnimi signali. To smo na sliki primerno označili s prekinjeno

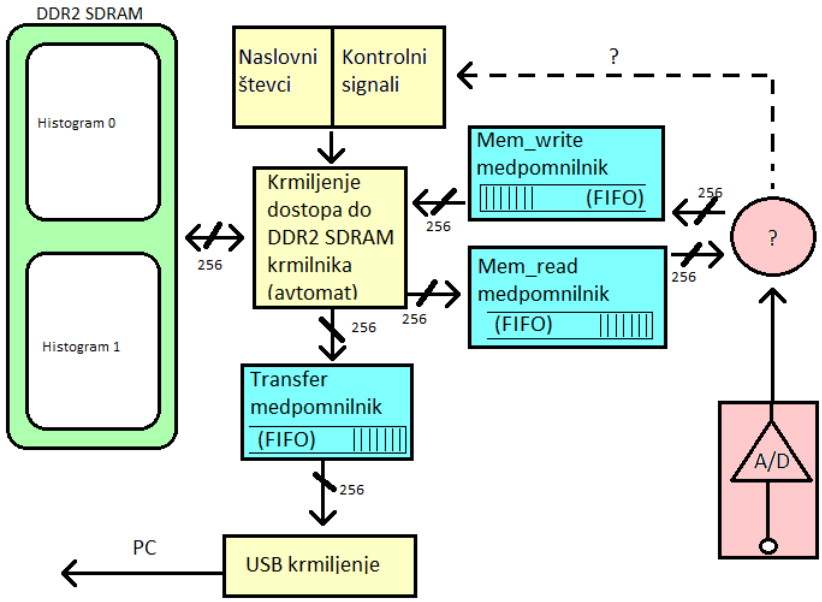


Slika 4.6: Stanja avtomata za prenos podatkov v USB medpomnilnik

črto, medtem ko je tok podatkov prikazan s polno črto puščice. Prav tako smo označili širino podatkovnih vodil.

Sedaj se lahko lotimo načrtovanja še mankajočega dela arhitekture. V eni urini periodi lahko z Mem_read medpomnilnika preberemo osem števcev histograma oziroma dva števca na kanal. Če želimo imeti časovno velikost predala histograma 6.4 ns, potem moramo pri frekvenci vzorčenja 1,25GHz vzeti osem vzorcev. Teh osem vzorcev moramo nato primerjati z mejno vrednostjo in pod pogojem, da je kakšen vzorec večji od te vrednosti, povečati vrednost ustreznega števca predala histograma. Dodamo še histerezno zanko in postavimo pogoj, da števec popravimo samo takrat, ko je rezultat primerjave prejšnjih osmih vzorcev enak 0 in rezultat primerjave trenutnih osmih vzorcev enak 1. S tem bomo zajeli samo prve prehode preko mejne vrednosti.

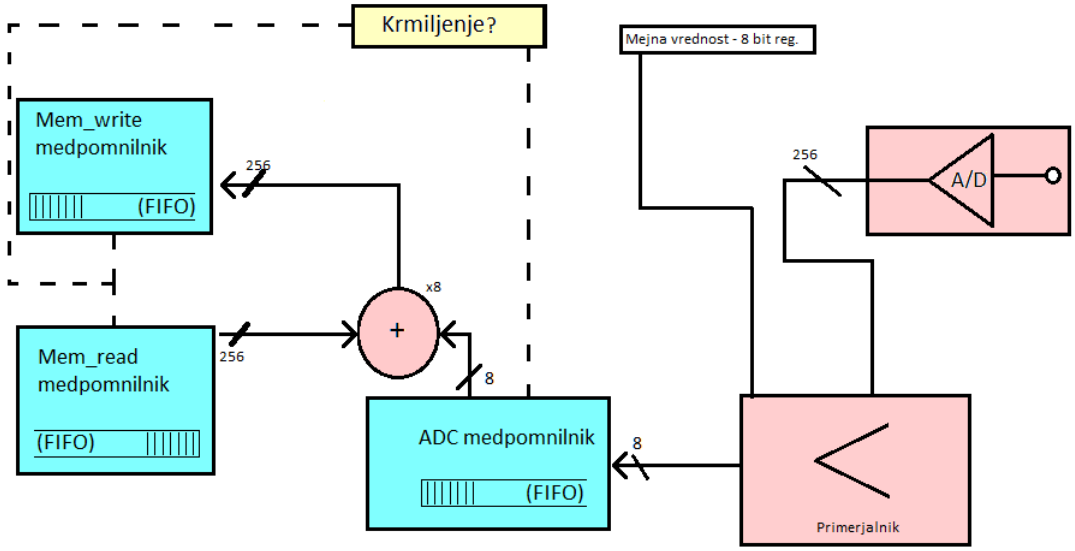
Rabimo torej dve urini periodi, da dobimo rezultate primerjav z mejno vrednostjo, medtem ko potrebno količino števcev histograma, ki čakajo v Mem_read medpomnilniku, preberemo v eni urini periodi. Vendar pa bomo ob začetku novega cikla histograma omenjeni medpomnilnik ponastavili in bomo morali nekaj urinih period čakati na podatke. To čakanje nam pogojuje uvedbo novega ADC medpomnilnika, ki implementira FIFO čakalno vrsto in bo širine 8 bitov, v njega pa bomo shranjevali rezultate primerjav za osem števcev. Torej



Slika 4.7: Pregled do sedaj načrtovane arhitekture

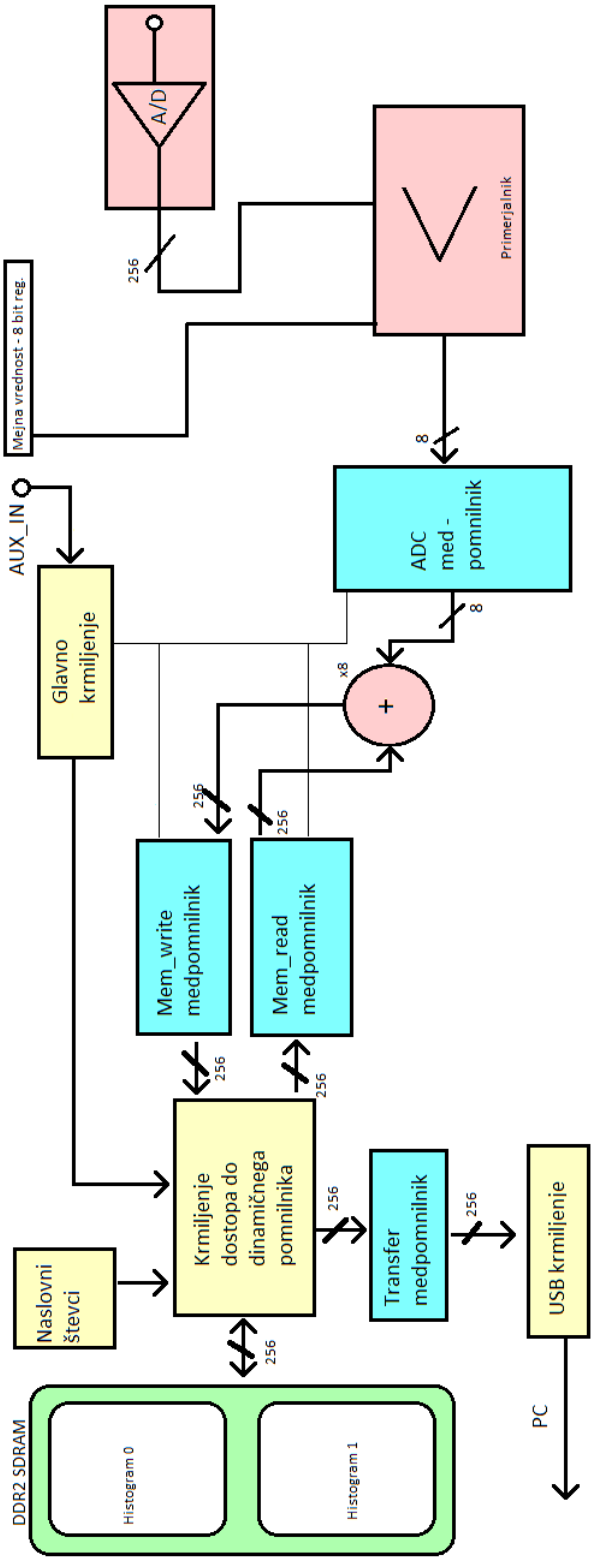
bomo v ta medpomnilnik pisali vsako drugo urino periodo, iz njega pa bomo brali, ko bodo na voljo podatki v obeh medpomnilnikih (ADC in Mem_read). Ko bomo prebrali podatke iz obeh medpomnilnikov, bomo opravili primerno seštevanje in rezultat zapisali v Mem_write medpomnilnik. Arhitekturo tega razmišljanja smo prikazali na sliki 4.8. S prekinjeno črto smo označili povezave, ki jih še moramo dopolniti do celotne arhitekture sistema. Preden pa lahko dokončamo našo arhitekturo, moramo premisliti, kaj se bo dogajalo ob koncu cikla histograma oziroma ob pričetku novega cikla.

Branje in pisanje podatkov v dinamični pomnilnik poteka v večjih kosih (ang. bursts), z enim ukazom za branje dostopamo do osmih zaporednih 128-bitnih besed oziroma do 1024 bitov. Glede na naše znanje o časovnih zakasnitvah dostopa do podatkov pri DDR2 SDRAM-u pa hkrati dopustimo možnost, da bomo zaporedno opravili več takšnih operacij in s tem še zmanjšali čas dostopa do podatkov. Pisanje podatkov seveda opravljamo v enako velikih kosih podatkov, kot opravljamo pisanje. Vendar pa velikost histograma ni nujno večkratnik velikosti teh blokov, kar pomeni, da moramo upoštevati posebnosti ob koncu cikla histograma. Ko zaznamo pozitivno fronto na vhodu sinhronizacijskega signala, moramo najprej pogledati, koliko je še preostalih podatkov v ADC medpomnilniku. Ti podatki namreč pripadajo še "staremu" ciklu. Zatem moramo še izračunati, koliko podatkov moramo samo prenesti iz Mem_read v



Slika 4.8: Popravljena arhitektura s seštevalniki in primerjalnim blokom

Mem_write medpomnilnik, da bi zadostili pogoju večkratnosti velikosti bloka podatkov za pisanje v DDR2 SDRAM in toliko podatkov tudi prenesemo. Ko zaključimo s tem prenosom, sporočimo avtomatu za dostop do pomnilnika, da smo končali s trenutnim ciklom in koliko podatkov smo zapisali v Mem_write medpomnilnik. Avtomat se na to odzove tako, da ponastavi Mem_read medpomnilnik, prebere primerno količino podatkov iz Mem_write medpomnilnika, ponastavi naslovne števce in primerno krmili signal, ki daje informacijo, da je končal z izvajanjem operacij ob koncu cikla histograma (slika 5.3). Ob potrditvi, da je avtomat pripravljen na gradnjo novega cikla histograma, lahko nadaljujemo tudi v tej (ADC) časovni domeni. Celotno arhitekturo lahko vidimo na sliki 4.9. Na sliki nismo prikazali vseh kontrolnih signalov, kar pravzaprav za razumevanje ni potrebno. Krmiljenje avtomata in ostali del krmiljenja delujeta v različnih časovnih domenah, zato smo prikazali signale za pisanje in branje medpomnilnikov.



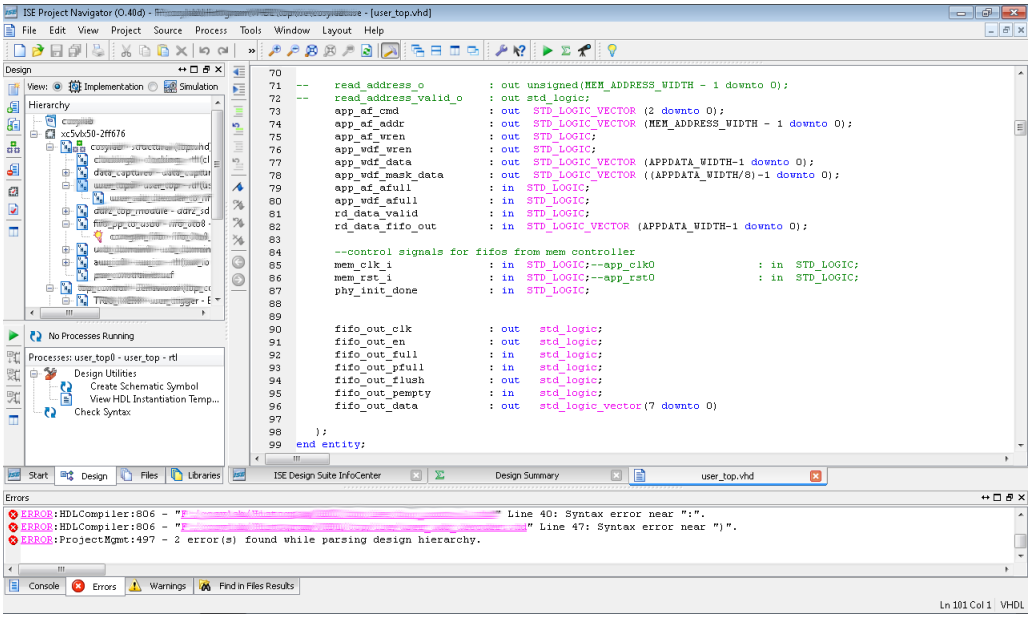
Slika 4.9: Končna arhitektura

Poglavje 5

Implementacija

Implementacija je proces prenašanja načrtovane arhitekture v obliko, primerno za fizično izvedbo, v našem primeru za programiranje. Kot smo že omenili, programiramo v jeziku VHDL (ang. VHSIC hardware description language). Pravila programiranja v VHDL nam narekujejo, da moramo oblikovati funkcijsko zaključene bloke, imenovane moduli. Moduli so sestavljeni z vhodnih in izhodnih signalov in notranje arhitekture, ki manipulira s temi signali. Moduli so lahko zelo majhni, npr. števci, lahko pa naredimo tudi večje, npr. modul za krmiljenje. Kako se odločimo, je popolnoma odvisno od nas, pogojeno pa je s praksami, s katerimi se srečujemo v okviru našega dela. V pričujočem poglavju se bomo osredotočili predvsem na sam proces implementacije in težave, ki smo jih imeli pri implementaciji.

Implementacijo pravzaprav pričnemo tako, da preučimo naše razvojno okolje. Kot že rečeno, bomo uporabljali Xilinx ISE razvojno orodje, posnetek zaslonske slike pa lahko vidimo na sliki 5.1. Orodje nam omogoča sintezo naše kode in generiranja "bitstream" datoteke, torej vse potrebno na tej stopnji implementacije. Na tem mestu je smiselno povedati malo več o samem postopku od programiranja do delujočega FPGA vezja. Ko sprogramiramo funkcijsko zaključen del, poženemo proces, imenovan sinteza. Sinteza poskrbi, da se sprogramiran modul sestavi kot skupina konfiguriranih logičnih blokov z opisano funkcionalnostjo. Rezultat sinteze je odvisen od tega, kako dobro smo funkcionalnost sprogramirali, tako da moramo med programiranjem zmeraj razmišljati o tem rezultatu. Postopku sinteze sledi postopek implementacije, ki pravzaprav izbere, kateri fizični bloki bodo izbrani na našem programabilnem čipu in po katerih povezavah bodo med seboj povezani. Tega procesa implementacije ne smemo zamenjevati s splošno implementacijo, kot jo opisujemo v tem poglavju. Prejšnja opisa sinteze in implementacije nista povsem



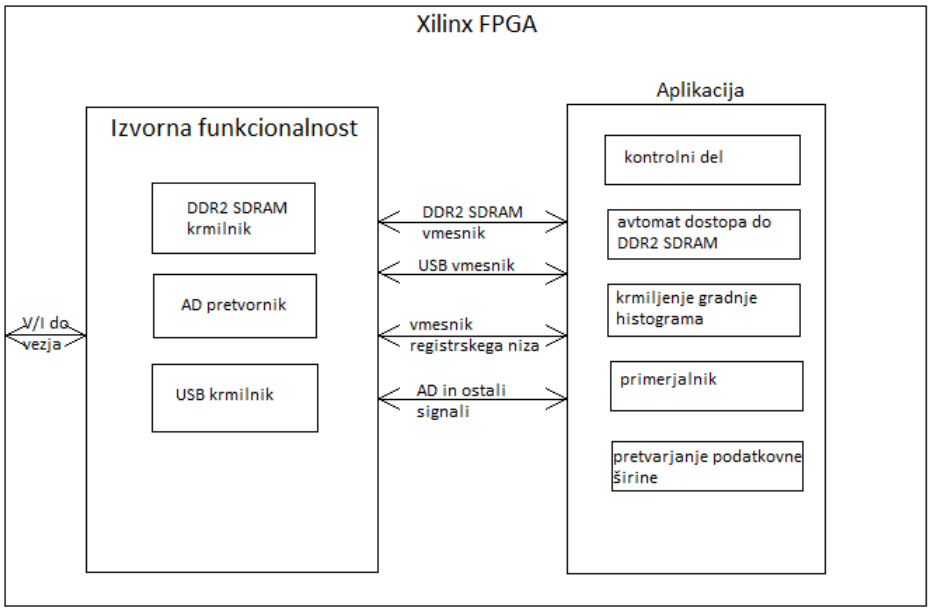
Slika 5.1: Posnetek zaslonske slike ob uporabi razvojnega okolja Xilinx ISE

natančna, vendar sta za razumevanje razlike med tema dvema procesoma dovolj. Produkt procesa implementacije je "bitstream" datoteka. To je datoteka, ki vsebuje konfiguracijo logičnih blokov in povezav FPGA čipa, zato posledično s to datoteko nastavimo čip.

Sedaj se lahko posvetimo namestitvi našega FPGA vezja s hitrim AD pretvornikom. Na računalniku najprej namestimo gonilnik v primerno mapo, gcc prevajalnik in poženemo osnovno kodo. Govorimo o osnovni kodi, ki omogoča računalniku krmiljenje tega vezja oz. klicanje v gonilnik vgrajenih funkcij. Ta koda je namenjena samo demonstraciji uporabe osnovnih funkcij, kot je nastavljanje kanalov, prenašanje bloka podatkov proti računalniku, nameščanje naše lastne "bitstream" datoteke, branje in pisanje v registrski niz in podobno. Na tem nivoju moramo rešiti vse probleme, ki bi lahko bili izključno napaka že izdelanega FPGA vezja s hitrim AD pretvornikom. Če že na začetku ne preizkusimo vseh dokumentiranih lastnosti vezja, če se ne spoznamo z delovanjem naprave in jo usposobimo, potem se težave samo prenašajo naprej. Na koncu namreč potrebujemo delujoč sistem in prej, kot rešimo vse težave, lažje bomo zaključili projekt. Jezik VHDL je še posebej zahteven pri razhroščevanju (ang. debugging) izvorne kode in potrebno je ogromno izkušenj, da uspešno zaključimo projekt, zato vse težave rešujemo sproti, če je to le mogoče.

5.1 Pregled implementacije

Kot že rečeno, arhitekturo najprej "razrežemo" na manjše module. Naš razrez je viden na sliki 5.2.



Slika 5.2: Pregled modulov in signalnih linij v najvišjem modulu

Sistem smo razdelili na šest večjih modulov:

- Kontrolni modul uporabimo za krmiljenje na najvišjem nivoju. Namenski krmilni modul je smiseln zato, ker na enem mestu združimo generiranje raznih krmilnih signalov. Vse funkcionalnosti, kot sta prenašanje in začetek gradnje histograma, moramo namreč krmiliti s primernimi signali.

- Modul dostopa do krmilnika dinamičnega pomnilnika je naš modul, ki vsebuje končni avtomat za dostop do dinamičnega pomnilnika. Prav tako potrebujemo še naslovne števce, ki jih lahko realiziramo v tem delu.

- Modul za krmiljenje gradnje histograma potrebujemo, da uskladimo dostop do različnih medpomnilnikov. Poskrbeti moramo za to, da bomo rezultate primerjav vzorcev z mejno vrednostjo prišteli k pravemu števcu, zato lahko to združimo v en modul.

- Primerjanje vzorcev z mejno vrednostjo in zapisovanje rezultatov v medpomnilnik lahko realiziramo v ločenem modulu. Morda bomo kdaj v prihodnosti še dodali kakšno funkcionalnost, zato je smiselno narediti ta modul tako enostaven.

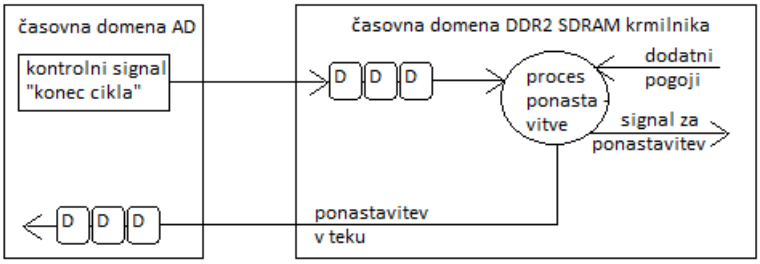
- Za avtomat pretvarjanja podatkovne širine uporabimo ločen modul. Modul je enostaven za implementacijo in precej neodvisen od vseh ostalih.

- Potrebujemo tudi glavni modul, v katerem imamo instancirane in med seboj povezane vse druge module. Prav tako bomo v tem modulu realizirali seštevalnike. Pravzaprav je takšen modul realiziran samo zaradi preglednosti celotnega sistema.

Lahko rečemo, da smo na tej točki razrešili vse večje probleme pri načrtovanju naše aplikacije. Sedaj moramo samo še implementirati posamezne module, poskrbeti za pravilno komunikacijo med njimi in končno preizkusiti delovanje aplikacije v praksi. Kakšen bo razrez naših modulov po funkcionalnosti je pravzaprav nepomembno in je namenjeno samo preglednosti in nadaljnji uporabnosti kode. Najpomembnejša je načrtovana arhitektura, kako pa bomo to arhitekturo zapisali in strukturirali v jeziku VHDL, pa je odločitev programerja.

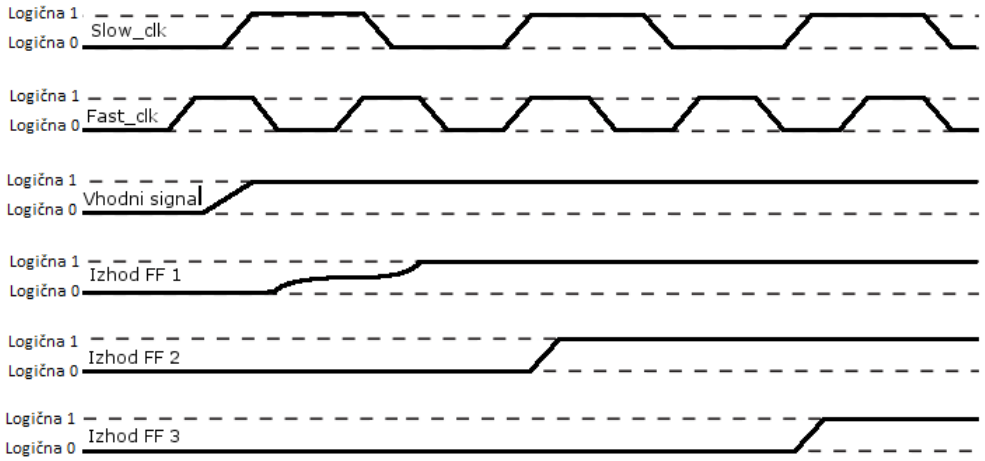
Implementacije modulov ne bomo posebej predstavljali, pogledali si bomo le specifičen problem, ki smo ga težko rešili pri implementaciji, ker nismo dovolj podrobno premislili rešitve pri načrtovanju arhitekture. Pri načrtovanju arhitekture smo se osredotočili na tok podatkov, nismo pa načrtovali krmilnega vezja. V nadaljevanju bomo osvetlili del tega problema in podali delno sliko implementacije.

Posebno pozornost bomo posvetili krmiljenju, ki skrbi za ponastavitev ob koncu gradnje enega cikla histograma. Ko nam sinhronizacijski signal poda informacijo, da moramo pričeti z gradnjo novega cikla histograma, moramo najprej zaključiti z gradnjo trenutnega cikla. To pomeni, da moramo najprej uskladiti količino prebranih in zapisanih podatkov v medpomnilnike s količino podatkov, ki jih obravnavamo v enem ciklu dostopa do dinamičnega pomnilnika. Velikost histograma po vsej verjetnosti ni večkratnik dolžine bralnega ali pisalnega dostopa do DDR2 SDRAM-a, ki smo jo implementirali z avtomatom za dostop do dinamičnega pomnilnika. To pomeni, da moramo nekaj števcev histograma samo prebrati iz Mem_read medpomnilnika in jih nespremenjene zapisati v Mem_write medpomnilnik. Nato moramo ponastaviti Mem_read medpomnilnik in ponastaviti naslovne števce. Vsi ti izpolnjeni pogoji nam omogočijo, da lahko pričnemo z gradnjo novega cikla histograma. Na sliki 5.3



Slika 5.3: Realizacija logike za ponastavitev ob koncu cikla histograma

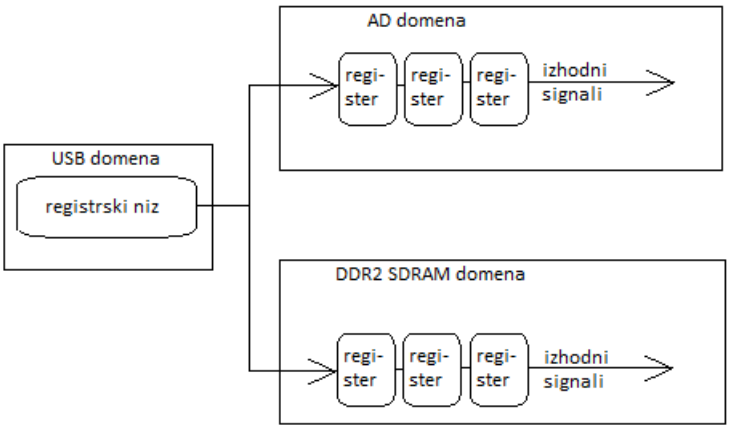
smo prikazali, kako uskladimo opisano ponastavitev v obeh časovnih domenah, kjer poteka.



Slika 5.4: Nivoji signalov pri prečkanju časovne domene

Na sliki 5.4 smo prikazali nivoje signalov pri prečkanju časovne domene, kot smo to opisali v prejšnjem odstavku. Vidimo dva različna urina signala, signal, ki prečka časovno domeno iz hitrejše v počasnejšo in tri pomnilne celice (ang. flip-flop). Vidimo, da se je prva pomnilna celica postavila v metastabilno stanje[4][5], o katerem bomo več povedali v podpoglavju o težavah pri implementaciji. Zaradi tega problema je potrebno signal pri prečkanju časovne

domene najprej sinhronizirati.



Slika 5.5: Prikaz uporabe registrskega niza pri različnih časovnih domenah

Na sliki 5.5 smo prikazali sinhronizacijo registrskega niza z dvema različnima urama, kjer je sam registrski niz na lastni časovni domeni. Ta problem smo si izbrali zato, da prikažemo, kaj je pravzaprav največja težavnost implementacije našega sistema. Za vse signale, ki jih pri implementaciji uporabljamo, moramo vedeti, iz katere časovne domene prihajajo in v kateri časovni domeni jih uporabimo.

Ostalih funkcionalnosti ne bomo tako orisali, saj smo jih do potankosti razčlenili že v poglavju o načrtovanju arhitekture. Pomembno je, da se držimo načrtovane arhitekture in skrbimo za pravilno postavljanje kontrolnih signalov, sploh pri prečkanju časovnih domen.

5.2 Težave pri implementaciji

Kodo, zapisano v jeziku VHDL, je težko razhroščevati, saj je naše preverjanje sestavljeno predvsem iz opazovanja odziva signalov izhoda modulov na simulatorju. Naš vpogled v napravo in opazovanje signalov sta omejena. Pri razvojnih ploščah imamo na voljo JTAG priključek, kjer lahko s programom ChipScope opazujemo, kakšni so dejanski nivoji signalov v nekem programabilnem čipu. V našem primeru smo za to prikrajšani, saj razvijamo aplikacijo

na že končnem izdelku – FPGA vezju s hitrim AD pretvornikom, in nam ChipScope ni na voljo. Tako je naš sistem posledično črna škatla, kjer nastavljamo vhodne signale in opazujemo izhodne, ne da bi imeli vpogled v dogajanje znotraj naše aplikacije. Pravilnost delovanja lahko ugotavljamo samo glede na odziv sistema. To naredi odpravljanje napak izjemno časovno potratno in zato problemom, povezanim z razhoščevanjem, posvetimo posebno podpoglavje.

5.2.1 Funkcijsko razumevanje izvorne kode FPGA vezja s hitrim AD pretvornikom

Kot že omenjeno, dobimo na začetku skupek kode, ki jo je sprogramiral nekdo drug. Če želimo s to kodo karkoli početi, potrebujemo kar dobršen del časa, da jo popolnoma razumemo. Ne samo, da nimamo nikakršnih grafov o izhodnih signalih modulov sistema, ampak tudi nimamo opisov o modulih in krmiljenju. Proces prečesavanja kode je zelo dolgotrajen, pomembno pa je, da ga opravimo kvalitetno. Če kakšnega dela kode ne preučimo, potem se nam lahko zgodi dvoje; bodisi bomo morali to storiti kasneje, bodisi lahko zaradi tega podvojimo dele arhitekture, ker ne vemo, da so že realizirani.

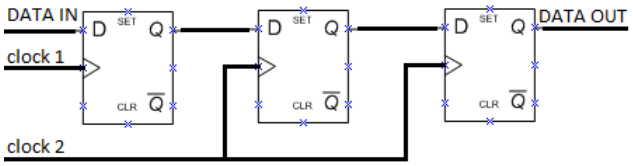
5.2.2 Komunikacija načrtovane funkcionalnosti z izvorno kodo

V prejšnjem podpodpoglavju smo govorili o tem, da moramo vedeti, kaj pravzaprav dela izvorna koda, ki smo jo dobili na začetku. Vendar pa to ni dovolj, če ne znamo te kode uporabiti, oziroma, če ne znajo naši moduli komunicirati z izvornimi moduli. Najlažje komuniciramo z nekim modulom, če imamo opis signalov, opisano funkcionalnost in priložen graf izhodnih signalov glede na vhodne signale. Na tem grafu se vidijo časovni zamiki ob izdaji ukazov in odzivi modula. Če teh grafov nimamo, moramo najprej funkcijsko razumeti kodo, potem pa še preučiti, kako se obnašajo izhodi modula, saj lahko samo tako komuniciramo z njim. Zgoraj omenjeno preučevanje kode modulov do nivoja signalov opravljamo vseskozi, saj moramo naše module ustrezno implementirati, da bodo pravilno komunicirali. Kot primer težav, ki se lahko pojavijo v tem primeru, je AD pretvornik. Omenili smo, da dobimo v eni urini periodi 256 bitov podatkov, ki pripadajo vzorcem, kjer je vsak vzorec širok 8 bitov. To nam predstavlja 32 vzorcev, vendar pa nam podatkovni tip in razporeditev nista nikjer pojasnjena. Ne vemo namreč, ali gre za zapis v dvojiškem komplementu, in ne vemo, kakšnega podatkovnega tipa so zajeti vzorci. Pri jeziku VHDL je v standardni uporabi več podatkovnih tipov in

hitro lahko pride do napak pri realizaciji seštevalnika ali primerjalnika, kar je posledica naše napačne uporabe podatkovnega tipa.

Prav tako je VHDL značilen jezik s stališča stila programiranja. Podjetja se zato držijo nekih dogovorjenih internih pravil pri programiranju, tako da je predajanje kode med programerji lažje in tudi koda hitreje razumljiva. Kadar kot programer dobimo kodo od nekoga izven podjetja, se hitro zgodi, da začnemo spreminjati naš stil in s tem tvorimo svojega. Pomembno je naše razumevanje, da so interna pravila v podjetju rezultat znanja vseh programerjev in da so postavljena zato, ker so preizkušena in delujejo. Moramo biti disciplinirani in se držati teh pravil, saj lahko drugače naletimo na težave, ki so posledica izključno tega, da nismo uveljavili internih pravil (npr. uporaba standardne knjižnice). Seveda bi moral programer poznati jezik, v katerem programira, vendar pa kot začetniki v programiranju v jeziku VHDL takoj na začetku ne moremo znati vsega.

5.2.3 Prečkanje različnih časovnih domen



Slika 5.6: Prenos signala z ene v drugo časovno domeno

V kolikor želimo module med seboj povezati s signali, morajo ti moduli vsi delovati z isto uro. Problem prečkanja časovnih domen se pojavi, ko želimo prenašati signale med dvema moduloma, kjer vsak deluje pri različni frekvenci urinega signala. Takrat se lahko zgodi, da želimo uporabiti signal, ki ni sinhroniziran z modulu lastno uro in v času preklopa ure napetostni nivo na opazovani liniji ni stabilen.

Do takšnih problemov torej pride, ko imamo opravka s komunikacijo med moduli, kjer uporabljamo različne ure, pri katerih ti moduli delujejo. Pri podatkovnih linijah smo ta problem elegantno rešili z uporabo hitrega pomnilnika, ki implementira FIFO čakalno vrsto, kjer je že Xilinx IP Core poskrbel za module, ki omogočajo delovanje bralnih in pisalnih vrat medpomnilnika pri različnih hitrostih ure. Vendar pa bomo med različnimi časovnimi domenami prenašali tudi kontrolne in naslovne signale. Tukaj se pojavi poseben problem,

imenovan metastabilno stanje[4][5], pri katerem pride do situacije, da vhodni signal pomnilne celice ob času preklopa ni logična "0" ali "1", ampak je nestabilen. To povzroči, da se tudi pomnilna celica ne postavi nujno kot je željeno, postavi se v t.i. metastabilno stanje, kar pomeni napačno delovanje. Primer takšnega stanja si lahko pogledamo na sliki 5.4, kjer je vhodni signal spremenil logično vrednost ravno ob času preklopa prve pomnilne celice. Pomnilna celica je v metastabilnem stanju relativno kratek čas, vendar dovolj dolgo, da s tem povzroči potencialno napačno delovanje. Problem rešimo tako, da vsak signal, ki prečka časovno domeno, vstopi v to domeno preko vsaj dveh zaporednih pomnilnih celic, kot vidimo na sliki 5.6. S tem poskrbimo, da zadnja pomnilna celica zagotovo ne bo v metastabilnem stanju, in lahko rečemo, da smo uspešno prečkali časovno domeno. Če prenašamo več signalov, moramo seveda zagotoviti, da vsi uporabijo enako število pomnilnih celic, samo orodje implementacije pa bo poskrbelo, da bodo signalne poti v čipu fizično enako dolge in s tem tudi preklopi enaki. Enako kot s kontrolnimi signali, naredimo tudi z naslovnim vodilom.

Primer tega problema smo opazili med razhoščevanjem, kjer smo s pomočjo enostavnega števca generirali podatke za vpis v enega od medpomnilnikov v sistemu, pisalna vrata medpomnilnika pa smo povezali z drugim signalom ure, tako da sta bili frekvenci različni. Končen rezultat branja so bili zamaknjeni podatki, ki so nakazovali na pokvarjene pomnilne celice dinamičnega pomnilnika ali na kakšen drug podoben problem. Zaradi tega moramo biti še posebej pazljivi pri uporabi istih signalov v različnih časovnih domenah.

Poglavje 6

Rezultati

Ko realiziramo naš sistem in zaključimo z razhroščevanjem, samo še preverimo pravilnost delovanja našega sistema. Seveda že vseskozi uporabljamo napravo imenovano funkcijski generator, ki nam generira signale, s katerimi preučujemo delovanje osciloskopa.

6.1 Predstavitev testnega okolja

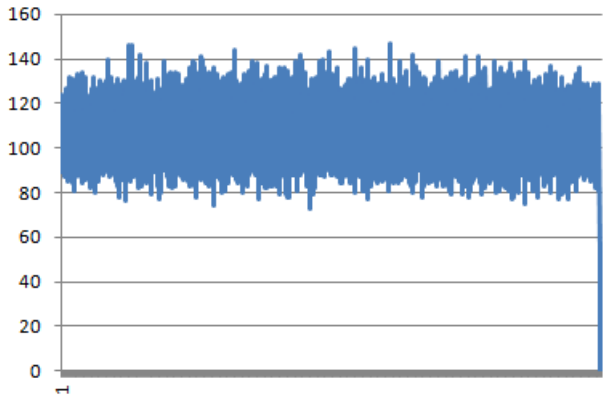
Slika 6.1: Funkcijski generator Rigol dg1022

Našega sistema ne moremo predstaviti pri delovanju v praktičnem okolju, saj bi za to potrebovali pospeševalnik delcev in poseben senzor. Zato takšne pogoje delovanja simuliramo v testnem okolju. Testno okolje je sestavljeno iz

našega osciloskopa in funkcijskega generatorja podjetja Rigol, model dg1022 (slika 6.1). Funkcijski generator nam generira dva signala; prvi je pravokotni signal točno določene frekvence in ga bomo uporabili kot sinhronizacijski signal. Drugi signal bomo generirali kot šum, pravokotni, ali sinusni signal, frekvenco pa bomo spreminjali. Kljub spreminjanju te frekvence bomo poskrbeli, da bo to večkratnik frekvence sinhronizacijskega signala. S tem bomo poskrbeli, da bodo vsi dogodki znotraj nekaj ločenih skupin predalov histograma. S pomočjo teh rezultatov bomo videli, ali naš sistem deluje pravilno in točno.

6.2 Predstavitev in analiza rezultatov

Najprej smo napisali kratko aplikacijo v jeziku C, ki nam je nastavila pravilne vrednosti vseh registrov na naši napravi. Tako je naša naprava pričela z gradnjo histograma. Ko smo po nekaj časa zahtevali prenos histograma, smo te podatke shranili v datoteko, da jih lahko grafično prikažemo. Zraven pridobimo tudi podatek o številu ciklov gradnje histograma. V vseh spodaj opisanih primerih smo histogram gradili približno 17800 ciklov.



Slika 6.2: Histogram pri šumu kot vhodni funkciji

Sedaj nastavimo prvi izhod funkcijskega generatorja na kvadratno funkcijo. Drugi izhod smo najprej nastavili na “noise”, torej na šum, kar bi moralo povzročiti približno enakomerne vrednosti histograma. Rezultat imamo na sliki 6.2, vidimo pa lahko, da imamo na koncu histograma “prazne” predale. Na grafu je to težko natančno opaziti, zato si poglejmo vrednosti predalov v

tabeli 6.1. Vidimo lahko, da so vrednosti v zadnjih predalih 0. To lahko pojasnimo z dejstvom, da velikost histograma ni nujno enaka večkratniku velikosti enkratnega dostopa do dinamičnega pomnilnika, kot smo jo opisali v poglavju o arhitekturi. Prav tako lahko opazimo, da je histogram monoton. Naš sklep je, da je rezultat pričakovan in delovanje aplikacije do sedaj pravilno.

zap. št. predala	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
vrednost predala	114	124	115	50	38	1	3	0	0

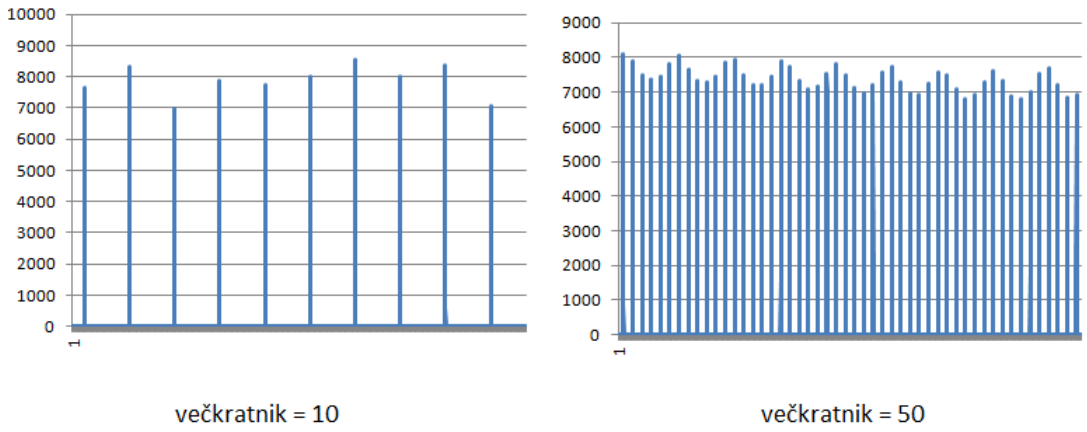
Tabela 6.1: Vrednosti izbranih zaporednih predalov ob koncu histograma na sliki 6.2

Nato bomo tudi drugi izhod nastavili na konkretno funkcijo, katere frekvenca bo večkratnik frekvence sinhronizacijskega signala. Najprej se odločimo za pravokotno funkcijo, prvič uporabimo frekvenco z večkratnikom 10, drugič pa z večkratnikom 50. Rezultat lahko vidimo na sliki 6.3. Če natančno pregledamo vrednosti histograma, lahko opazimo, da je število lokalnih maksimumov enako večkratniku med vhodnima frekvencama. To je pričakovano in potrjuje pravilnost delovanja naše aplikacije. Če natančno pregledamo vrednosti, lahko opazimo, da en takšen lokalni maksimum zavzema štiri predale. To lahko pripišemo nenatančnosti vhodnih signalov. Vsota vrednosti na območju nekega lokalnega maksimuma je enaka številu ciklov izvajanja histograma, kar nas pripelje do sklepa, da naša aplikacija deluje pravilno.

Št. maks.	predal 1	predal 2	predal 3	predal 4	predal 5	predal 6
1	0	2036	8094	6899	805	0
2	19	2157	7906	6824	928	0
3	2	2584	7506	6339	1403	0
4	0	2735	7364	6094	1641	0
5	0	2619	7438	6262	1515	0
6	0	2193	7813	6711	1117	0
7	0	1903	8073	7018	840	0

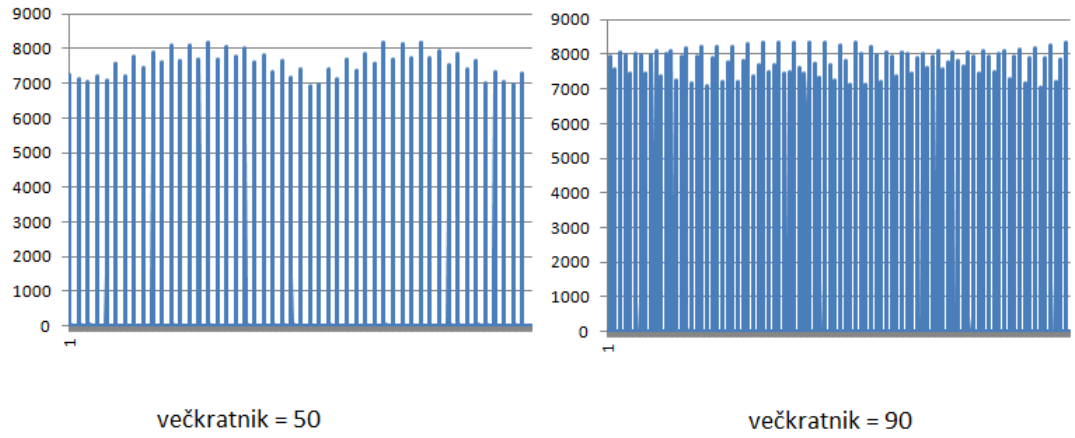
Tabela 6.2: Vrednosti predalov lokalnih maksimumov z drugega histograma na sliki 6.3

Opazimo lahko tudi zanimiv fenomen, ko tudi lokalne maksimalne vrednosti predalov predstavljajo nekakšen sinus. Točne vrednosti predalov na drugem grafu slike 6.3 lahko vidimo v tabeli 6.2. Pod številko maksimuma imamo v



Slika 6.3: Histogram pri kvadratni funkciji kot vhodni funkciji

mislih zaporedno številko lokalnega maksimuma, ki se pojavi na grafu. Pregled vrednosti v tabeli nam ne poda pojasnila za naša opazovanja, vidimo lahko samo, da so vrednosti različno razpršene. To zanimivo nihanje vrednosti lokalnih maksimumov lahko pripišemo več dejavnikom, npr. šumu zaradi napajanja, AD pretvorniku, itd.



Slika 6.4: Histogram pri sinusni funkciji kot vhodni funkciji

V zadnjem koraku bomo kot vhodno funkcijo preizkusili še sinusno funkcijo. Izbrali smo si frekvenci z večkratnikoma 50 in 90. Histograma lahko vidimo na sliki 6.4. Natančen pregled vrednosti histograma potrjuje naše dosedanje ugotovitve. Vsota vrednosti v lokalnih maksimumih je enaka številu

ciklov gradnje histograma in zavzema večinoma štiri predale s skoraj enako porazdelitvijo kot pri kvadratni funkciji. Izrazita sta dva predala, kamor pade skoraj 80 % vseh dogodkov, ostali pa v stranske predale. Tudi v tem primeru lahko opazimo zanimiv pojav šuma, kot v prejšnjem primeru.

Glede na vse podane rezultate in ugotovitve je naš sklep takšen, da načrtovana aplikacija za gradnjo histograma na FPGA vezju v realnem času deluje pravilno.

Poglavje 7

Zaključek

V diplomski nalogi smo uspešno načrtovali arhitekturo in implementirali aplikacijo za gradnjo histograma v realnem času na FPGA vezju. To smo potrdili z grafičnim prikazom rezultatov in njihovo analizo. Ugotovili smo, da je načrtovanje arhitekture pravzaprav zelo zapleten postopek, ki potrebuje tehtno razmišljanje in neprestano preverjanje smiselnosti predlaganih rešitev. Skozi postopek moramo sprejeti odločitve, za katere zgolj ugibamo, da so pravilne, saj bi natančna analiza trajala predolgo. Hkrati moramo predvsem zadostiti zahtevam naročnika aplikacije, s katerim smo se dogovorili za časovni rok izvedbe projekta. Najpomembnejši kriterij je namreč, da je naročnik z izdelkom zadovoljen, ne pa razvoj najboljše možne aplikacije z danimi resursi.

Pri pozornem preučevanju načrtovane arhitekture lahko ugotovimo, da obstaja en vir nenatančnosti v naši arhitekturi, ki lahko prispeva k zmanjšani točnosti histograma. Zaradi naše nenatančnosti lahko pride do zamika ene urine periode in posledično enega predala histograma. Hkrati se moramo zavedati, da tudi vsi vhodni signali vsebujejo šum in napake. Na primer pri digitalnem vhodnem signalu frekvence kroženja gruč ne moremo trditi, da je čas med poljubnima dvema zaporednima pozitivnima frontama povsem enak. Te naše sume o virih netočnosti in šuma pravzaprav potrjujejo naši rezultati. Postavili smo primerno testno okolje, a idealnih rezultatov, kjer bi bili vsi dogodki v enem predalu histograma, nismo dobili. S tem smo postavili izhodišče za nadaljnje izboljševanje arhitekture in sistema.

Programabilni čipi FPGA so danes zaradi svoje velikosti in hitrosti dovolj dobri za realizacijo zahtevnih, namenskih aplikacij. Naša diplomska naloga, realizirana arhitektura in še posebej predstavljeni rezultati to našo tezo potrjujejo.

Literatura

[1] Xilix, *Virtex-5 Family Overview*. DS100 (v5.0), 2009, http://www.xilinx.com/support/documentation/virtex-5.htm.

[2] Xilix, *Virtex-5 FPGA User Guide*. UG190 (v5.4), 2012, http://www.xilinx.com/support/documentation/virtex-5.htm.

[3] Xilix, *Memory Interface Solutions User Guide*. UG086 (v3.6), 2010, http://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf.

[4] http://www.asic-world.com/tidbits/metastablity.html, 25.7.2012

[5] Altera, *Understanding metastability in FPGAs*. wp-01082-1.2 (v1.2), 2009, http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf.