Tartu Ülikool
Loodus- ja täppisteaduste valdkond
Tehnoloogiainstituut

Oliver Oll

Õppeaine „Digitaalne loogika“ praktikumide täiendamine riistvaraliste paralleelülesannetega

Bakalaureusetöö (12 EAP)
Arvutitehnika eriala

Juhendaja:
MSc Margus Rosin

Tartu 2022

Resümee

Õppeaine „Digitaalne loogika“ praktikumide täiendamine riistvaraliste paralleelülesannetega

Bakalaureusetöö eesmärk oli tutvuda erinevate FPGA peal teostatavate riistvaraliselt paralleelsete ülesannetega ning luua nende põhjal Tartu Ülikooli õppeainesse „Digitaalne loogika“ (LOTI.05.041 ja LOTI.05.055) juurde praktikumiülesandeid. Töö käigus katsetas töö autor läbi erinevaid pilditöötluse võtteid ja analüüsis FPGA paralleelsust. Samuti valmis töö käigus kaks uut praktikumijuhendit, millest üks keskendus VGA graafika standardile ja teine Block RAM-i loomisele. Esimeses praktikumis kuvasid tudengid Basys3 arendusplaadi külge ühendatud monitorile läbi VGA-liidese monotoonse värvi, mida sai samalt arendusplaadilt liugnuppudega muuta. Teises praktikumis kirjutasid tudengid FPGA peal asuvale BRAM-i pildi, mille edastasid hiljem samast kohast lugedes ekraanile.

Tudengite tagasisidest selgus, et enamuse arvates olid praktikumid õpetlikud, vajalikud ja võiksid alles jääda ka edaspidi. Negatiivsete kommentaaridena toodi välja, et koodi kirjutamine arendusplaadile nõuab liigset täpsust ning mõningatel juhtudel jäi ülesande mõte ja sellega seonduv teooria osa selgusetuks.

CERCS: T111 Pilditehnika; T120 Süsteemitehnoloogia, arvutitehnoloogia; T170 Elektroonika; T190 Elektrotehnika

Mä**rksõnad**: Basys3, BRAM, Digitaalne loogika, FPGA, Pilditöötlus, VGA, VHDL

Abstract

Adding hardware-based parallel tasks to practical labs of course „Digital logic“

The objective of this Bachelor’s thesis was to learn about different hardware-based parallel tasks which could be carried out on FPGA, and to create new practical lab instructions on these tasks for the University of Tartu course „Digital logic“ (LOTI.05.041 and LOTI.05.055). The author of this thesis used various image processing methods and analysed parallel workflow of FPGA. In addition, two new practical labs were created for students. One of the labs concentrated on VGA graphics standard and displaying colors that were set as an input on development board Basys3. The second lab taught how to create a Block RAM component, write an image on it and display it.

Feedback revealed that most of the students found the practical labs to be educational, necessary, and that labs should remain as a part of the course „Digital logic“. On the ohter hand, writing the code required too much precision, in a few cases the point of labs did not become clear and basics of VGA and BRAM remained abstruse.

CERCS: T111 Imaging, image processing; T120 Systems engineering, computer technology; T170 Electronics; T190 Electrical engineering

Keywords: Basys3, BRAM, Digital logic, FPGA, Image processing, VGA, VHDL

Sisukord

Jooniste loetelu

Tabelite loetelu

Lühendid, konstandid, mõisted

BRAM (*Block Random Access Memory)* – muutmälu, mille andmed on füüsiliselt koondatud ühele mälualale.

FPGA (*Field Programmable Gate Array)* – programmeeritav ventiilmaatriks ehk kasutaja poolt programmeeritav integraalskeem.

GPS (*Global Positioning System)* – süsteem, mille abil on võimalik määrata asukohta Maal.

HDL (*Hardware Description Language)* – riistvarakirjelduse keel ehk programmeerimiskeel, mida kasutatakse elektroonikaskeemi funktsioonide kirjeldamiseks ja täitmiseks.

IP *Core* (*Intellectual Property Core)* – tarkvaraline moodul, mida saab kasutada teistes koodides.

LED (*Light-Emitting Diode)* – valgusdiood.

LUT (*Look-Up Table)* – väärtustetabel, millest saab võtta väärtusi vastavalt võtmele.

Pmod – ettevõtte Digilent, Inc. sisend-väljund arendusplaat.

RAM (*Random Access Memory)* – muutmälu, kuhu saab andmeid kirjutada ja millelt saab neid ka lugeda.

ROM (*Read-Only Memory)* – püsimälu, milles sisalduvaid andmeid ei saa muuta.

SoC (*System on a Chip)* – süsteemikiip ehk kiip, millesse on mahutatud terviklik lahendus (sisaldab nii arvutit kui ka tööks vajalikku lisaelektroonikat).

SPI (*Serial Peripheral Interface)* – liides, mille abil on võimalik andmeid vahetada.

UART (*Universal Asynchronous Receiver-Transmitter)* – liides, mille abil on võimalik andmeid vahetada.

VGA (*Video Graphics Array)* – videopildi edastamise standard ja liides.

VHDL (*Very High Speed Integrated Circuit* HDL) – üks enim kasutatud HDL-e.

1. Sissejuhatus

Töö autor läbis 2020/21. õppeaastal Tartu Ülikooli aine „Digitaalne loogika“, mille käigus ta õppis programmeerima Basys3 arendusplaadil olevat FPGA-d. Siiski ei kasutatud praktikumide käigus kõiki arendusplaadi võimalusi ega loodud suuremat paralleelsetest protsessidest koosnevat projekti. Selleks, et tulevased tudengid mõistaksid paremini paralleelsete protsesside olemust ja vajalikkust, otsustas töö autor luua uued praktikumijuhendid, mis toetaksid vastavaid õpiväljundeid. Samuti võiksid juhendid arendada tudengite oskust lugeda kasutusjuhendit ja õpetada rakendama sealt loetud teavet.

Juhendite loomiseks oli töö autoril tarvilik mõista, kuidas töötavad paralleelsed protsessid ja õppida ise kasutama arendusplaadi võimalusi. Seega keskendub bakalaureusetöö esimene pool töövahendite ja -võtetega tutvumisele ning teine pool nende praktikasse rakendamisele. Lisaks oli vajalik aru saada, kuidas tutvustada uut teemat tudengitele, leida sobiv ülesanne ja vajadusel neid suunata. Töö lõpp vaatleb praktikumijuhendite loomist ning analüüsib tehtud tööd, et edaspidi luua arusaadavamaid ja huvitavamaid õppematerjale.

2. Ülevaade probleemist

Tartu Ülikooli aine „Digitaalne loogika“ eesmärgiks on õpetada tudengitele riistvara kirjelduskeelt VHDL – selle olemust ja rakendamist FPGA-l. Lisaks vaatleb õppeaine digitaalsüsteemide sünteesimist, simuleerimist ja selleks vajalikke tööriistu. [1] Praktikumides on kasutusel Digilent, Inc.-i arendusplaat Basys3, millel on Artix-7 FPGA koos rohkete sisend-väljund seadmetega. Digilent, Inc. on ettevõte, mis pakub riist- ja tarkvaralisi tööriistu FPGA-d ja SoC-i kasutavatele süsteemidele. Näiteks loovad nad arendusplaate, millele on paigaldatud ettevõtte Xilinx, Inc.-i FPGA-d. Samuti ka arendusplaatidele ühendatavaid spetsiaalseid Pmod mooduleid. [2] Kuna FPGA tugevaimaks omaduseks on paralleelsed protsessid, kasutatakse seda laialdaselt näiteks pildi- ja signaalitöötluses. Seetõttu leiab töö autor, et ka praktikumides võiks mõnda sarnast teemat käsitleda. Bakalaureusetöö eesmärk on koostada tudengitele jõukohased paralleelsust demonstreerivaid praktikumi ülesanded.

2.1 Varasemad praktikumid

Aine „Digitaalne loogika“ 2020/21 õppeaasta praktikumide sisuks oli LED-ide ja liugnuppude kasutamine läbi reaaleluliste rakenduste (täissummaator, loendur, Mealy ja Moore’i olekumasinad jms). Lisaks õpetati ka *distributed* RAM-i looma (andmeid LUT-ide vahel ära jagama) ja kasutama (andmeid LUT-idest lugema ja nendesse kirjutama).

Varasemalt on ainesse 2019. lõputöö raames loodud praktikume Pmod moodulite kohta, mis näitasid tudengitele, kuidas kasutada erinevaid sisend-väljund seadmeid (nt audio ja GPS) ning nendega kaasnevaid kommunikatsiooni protokolle (nt UART ja SPI) [3]. Kahjuks edaspidi neid praktikume ei kasutatud, kuna pandeemia oludes polnud Tartu Ülikoolil piisavalt mooduleid, et kõikidele tudengitele neid korraga väljastada.

Eelnevalt on Eestis VHDL-i kirjutamist õpitud näiteks Tartu Ülikoolis aines „Crash Course in Digital Technology“ (tõlkes „Digitaalse tehnoloogia kiirkursus“), kus loodi selle aine eestvedaja - Ahti Laurissoni sõnul erinevaid projekte LED-ide vilgutamisest kuni ussimängu loomiseni. Samuti on riistvarakirjelduse keeli loetud TalTechis, kus on aine „Digitaalsüsteemid“, milles õpetatakse samuti VHDL-i, kuid sinna juurde rohkem koodi optimiseerimist ja aritmeetika- ning loogikaoperatsioone [4].

2.2 Pilditöötlus mujal maailmas

Pilditöötlust on varem tehtud FPGA peal näiteks 2019. aastal Hiinas Qingdao ülikoolis, et leida etteantud pildilt Sobeli algoritmi abil objektide servasid. Töö käigus võetakse värvilise pildi pikslid ning leitakse nii horisontaalses kui vertikaalses suunas asuvad tugevamad toonide muutused ehk objektide servad. Seejärel kuvatakse vastava piksli lõplik väärtus läbi VGA-liidese ekraanile. Loodud programmi on võimalik kasutada erinevates manussüsteemides. [5]

Sarnast tööd on tehtud ka 2014. aastal Austraalias Macquarie ülikoolis, kus kirjutati valmis sama algoritmi kasutav programm, mis loeb sisse reaalajalise videopildi, töötleb seda, määrab objektide kauguse kaamerast ning lõpuks väljastab ka kogu info ekraanile. Töö eesmärgiks oli luua aluskood süsteemile, mida võiks kasutada edaspidi inimese silmaiirise või näo tuvastamiseks. [6]

3 Metoodika

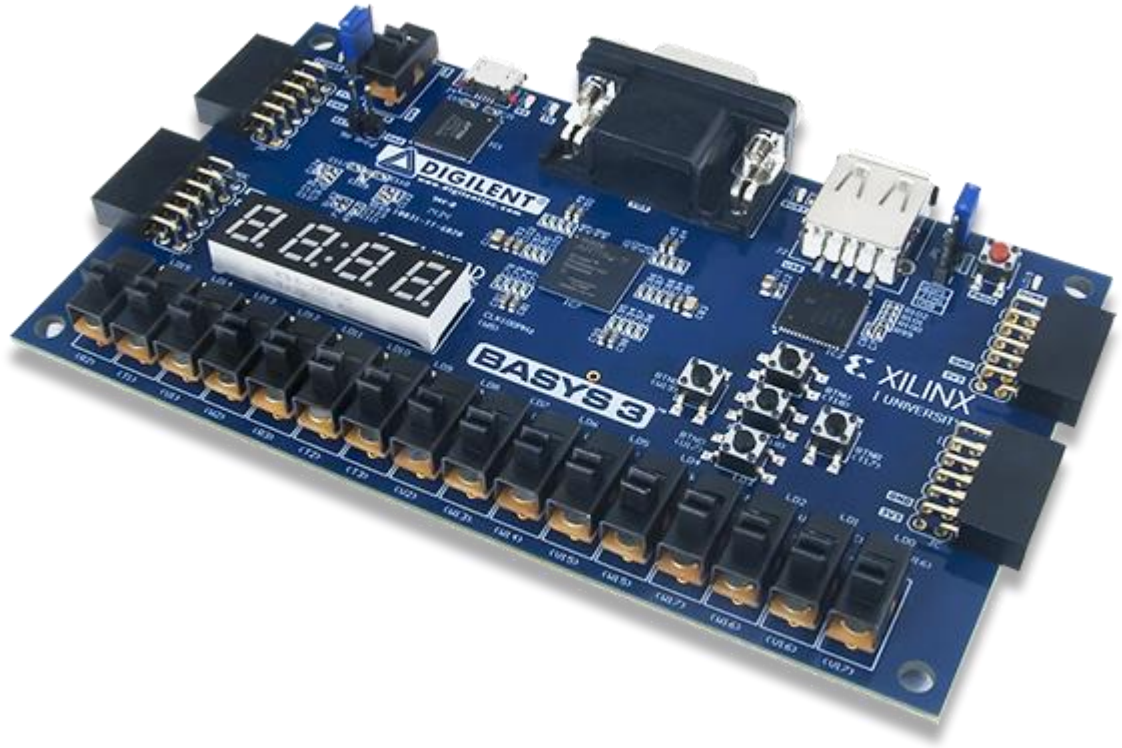
FPGA on väikestest programmeeritavatest loogikaplokkidest koosnev pooljuhtseade, mille eesmärk ja töökäik sõltub nende plokkide seadistusest [7]. Loogikaplokid koosnevad *slice*'idest, mille omakorda moodustavad *flip-flop*'id (elektroonika komponendid, mis muudavad oma väljundit sisendi(te) muutusel), LUT-id, multiplekserid (elektroonika komponendid, mis edastavad ühe valitud sisendsignaali väljundisse) ning matemaatilise ülekande loogikaplokk. FPGA suudab lahendada erinevaid ülesandeid paralleelselt, kuna töö jaotatakse ära loogikaplokkide vahel. [8] Tänapäeval kasutatakse FPGA-sid militaartööstuses, koduelektroonikas, andmetöötluses, meditsiinis, pildi- ja videotöötluses, telekommunikatsioonis ning paljudes muudes valdkondades [7].

HDL on programmeerimiskeel, mida kasutatakse riistvara töökäigu määramiseks. FPGA-de programmeerimiseks kasutatakse kahte enamlevinud HDL-i: Verilog ja VHDL [6].
FPGA-de suurimad tootjad maailmas on Intel Corporation, Xilinx, Inc., Lattice Semiconductor Corporation, Microchip Technology Inc. ja Gowin Semiconductor Corporation [9]. Bakalaureusetöös keskendub autor Xilinx, Inc.-ile ning tema tööriistadele, sest aine „Digitaalne loogika" praktikumides kasutataval Digilent, Inc.-i arendusplaadil on Xilinx, Inc.-i FPGA. Lisaks FPGA-de tootmisele, on Xilinx, Inc. ettevõte, mis ka lõi FPGA-d. Samuti pakub Xilinx, Inc. erinevaid tarkvarasid ja tööriistu tehnoloogia arendamiseks nii masinõppes, andmeanalüüsis kui video- ja pilditöötluses [10].

Vivado on Xilinx, Inc.-i tarkvara, mille abil saab luua süsteemi disaini, sünteesi, implementatsiooni ja simulatsiooni [11]. Samuti loob Vivado HDL-i põhjal riistvara jaoks loetava bitijada ning kirjutab selle FPGA-sse [12].

3.1 Basys3

Basys3 on Digilent, Inc-i loodud arendusplaat, mille peal on programmeeritav Xilinx, Inc.-i Artix-7 FPGA [12]. Artix-7 sees on 215000 loogikaplokki (igaüks sisaldab nelja LUT-i ja kaheksat *flip-flop*'i) ja 740 *slice*'i (igaüks sisaldab summaatorit arvude liitmiseks, korrutit arvude korrutamiseks ning registrit vahevastuste salvestamiseks). FPGA-l on ka 500 sisend-väljund viiku, mis suudavad andmeid vastu võtta ja välja saata kiirusega 1866 Mb/s, ning mälu, kuhu mahub maksimaalselt 13 Mb infot. [8] Lisaks FPGA-le on plaadil kasutajaga suhtlemiseks 16 liugnuppu, 5 surunuppu, 16 LED-i, 4-numbriline 7-segmendiline kuvar, 4 Pmod pesa, USB- ja VGA-liides [12] [13].



Joonis 1. Basys3 arendusplaat.

3.2 Digitaalse pildi töötlus

Kõik digitaalsed pildid koosnevad kindla asukohaga pikslitest, millel on arvulised väärtused. Pilditöötlus on meetod, mille abil muudetakse olemasolevat pilti, et seda täiustada mingi kindla eesmärgi saavutamiseks. [14] Mustvalgete piltide pikslitel piisab ühest väärtusest, mis määrab ära, kui hele või tume vastav piksel on. Kui vaadelda värvilisi pilte, siis peab iga piksel sisaldama erinevate värvitoonide infot, mistõttu ei piisa enam ühest arvust. Enamik värvusruume on 3-dimensionaalsed ehk värvust on võimalik edasi anda kolme väärtusega. Tuntuimad värvusruumid on RGB (*Red*, *Green*, *Blue*), kus värvuse saamiseks antakse edasi punase, rohelise ja sinise tooni intensiivsus, ning HSV (*Hue*, *Saturation*, *Value*), kus *Hue* määrab värvitooni, *Saturation* tooni puhtuse (kas toon on puhas või pigem kaldub värvusetu halli poole) ja *Value* tooni heleduse. [15]

Pilditöötluses kasutatakse kerneleid, mis on suuruselt võrdsete paarituarvuliste ridade ja veergudega maatriks. Kernel peab mõõtmetelt kindlasti olema väiksem või võrdne pildi lühema servaga ning tema elementideks on arvulised väärtused. Kerneli eesmärgiks on n-ö asetada ta pildile ja vaadata, millised pildi pikslid jäävad „akna“ (kerneli suuruse maatriksi) sisse. Seejärel määravad „aknasse“ sattuvate pildi pikslite olulisuse ära nendega vastavuses olevad kerneli arvulised väärtused ehk

vastavate pikslite kaalud. Taolist kernelit üle terve pildi kasutades (nii et kerneli keskmine element satub ühe korra igale pildi pikslile) on võimalik pilti töödelda. [16]

Lihtsaima näitena pilditöötlusest võib välja tuua hägustamise, mille käigus kaovad pildilt teravad servad ja objektid muutuvad hägusemaks. Sellise efekti saavutmiseks väärtustatakse kõik kerneli elemendid võrdselt tema elementide arvu pöördarvuga (Joonis 2) ehk kerneli elementide summa on 1. Kui asetada kerneli keskpunkt mingile pildi pikslile, siis leitakse selle piksli uus väärtus, korrutades kõik „aknasse“ jäävad kerneli elemendid nendele vastavate pildi pikslitega ning liites need korrutised kokku. Teisisõnu määratakse pildi iga piksli uueks väärtuseks teda ümbritsevate (sõltub kerneli suurusest) pikslite keskmine väärtus. [16]

Teiseks näiteks sobib Sobeli algoritm, mille abil leitakse pildilt objektide servasid. Sobeli algoritm jaguneb kaheks, millest üks leiab vertikaalseid servasid ja teine horisontaalseid servasid. Vertikaalsete servade leidmiseks on kerneli keskmine veerg väärtustatud 0-dega, parem- ja vasakpoolsed veerud sümmeetriliselt keskmise veeruga väärtustatud vastandarvudega, kusjuures keskmisele reale lähedasemad väärtused on suuremad (Joonis 3). Sellise kerneli ülesandeks on võrrelda ühe pildi pikslist paremale ja vasakule jäävaid piksleid ning leida, kas nende vahel on suuri erinevusi. Kui toonide vahe on etteantud lävendist suurem, on tegu vertikaalse servaga. Sarnaselt töötab horisontaalsete servade leidmine. Kõikide servade leidmiseks on vaja kombineerida mõlemad kernelid. [5]

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Joonis 2. Näide hägustamisel kasutatavast kernelist.

-1	0	1
-2	0	2
-1	0	1

Joonis 3. Näide vertikaalsete servade leidmisel kasutatavast kernelist.

3.3 Mälu

Xilinx, Inc. Vivado tarkvara sisaldab erinevaid IP Core’isid ehk mooduleid, mis kujutavad valmiskirjutatud programmi, mida saab hõlpsasti enda koodis kasutada. Kindla mooduli kasutamiseks tuleb Vivados luua graafiliselt vastav komponent, mille sisendisse saab hiljem saata

signaale ja mille väljundist signaale lugeda. [17] Näiteks leidub loendurit kujutav IP Core, mis loendab nii kasvavalt kui ka kahanevalt ja millega koos on võimalik kasutada väljundsignaali, mis kindlal lävendil muudab oma väärtust [18]. Samuti saab programmides rakendada DisplayPort videostandardi liidese IP Core'i, mis võimaldab saata ning vastu võtta videopilti [19].

Block Memory Generatori abil on võimalik luua erinevaid mälukomponente [20] - *Single Port* RAM, *Simple Dual Port* RAM, *True Dual Port* RAM, *Single Port* ROM, *Dual Port* ROM. Kõikide nende mäluelementide loomisel tuleb määrata nende suurus. Kuna mäluelementi võib ette kujutada maatriksina, mille igaks elemendiks on üks bitt, saame vaadelda tema suurust ridade ja veergude abil. Ridade arv määrab, kui palju on vaja kasutada mäluaadresse ja veergude arv määrab, mitmebitised vektorid asuvad igal real. [21] Suurim võimalik rea laius on 72 bitti. Maksimaalselt mahutab Artix-7 BRAM-idesse 1800 kB andmeid. [8] Kõiki lugemisi ja kirjutamisi teostatakse etteantud taktsignaali tõusval frondil [21], mis tähistab olukorda, kus sisendpinge tõuseb 0-st voldist vahemikku 2,1-3,9 volti [22].

Single Port RAM-i puhul kasutatakse järgnevaid signaale:

- DIN (*Data In*) – vektor, mis kirjutatakse mäluelemendi ADDR reale, kui WE on kõrges olekus.
- ADDR (*Address*) – väärtus, mis määrab, milliselt mälupesalt ehk aadressilt loetakse DOUT vektori väärtus, kui WE on madalas olekus, või millisele reale kirjutatakse DIN vektori väärtus, kui WE on kõrges olekus.
- WE (*Write Enable*) – bitt, mis määrab, kas ADDR reale kirjutatakse väärtus DIN või loetakse DOUT vektorisse väärtus ADDR realt.
- CLK (*Clock*) – bitt, mis kujutab taktsignaali ning mille tõusval frondil toimuvad kõik mäluelemendisisesed protsessid.
- DOUT (*Data Out*) – vektor, millelt saab lugeda ADDR rea väärtust, kui WE on madalas olekus.

Kirjutamisel kasutatakse režiime, kus vastavalt ADDR väärtusele muudetakse ühte mäluelemendi rida, kuid DOUT väärtus muutub erinevalt.

- *Write First* – vaikimisi režiim, mille käigus määratakse DIN väärtus koheselt ka DOUT vektorile.
- *Read First* – režiim, mille käigus määratakse ülekirjutatav ADDR rea väärtus DOUT vektorile.
- *No Change* – režiim, mille käigus DOUT vektorit ei muudeta, mistõttu muutub see ainult siis kui teostatakse lugemisoperatsiooni.

Dual Port RAM-ide puhul on kasutusel kaks erinevat porti, mistõttu saab nii üheaegselt lugeda kahte erinevat mälupesa, kirjutada kahele erinevale mälupesale kui ka lugeda ühte mälupesa ja kirjutada teisele mälupesale. *Dual Port* RAM-id jagunevad kaheks: *Simple Dual Port* RAM ja *True Dual Port* RAM. *True Dual Port* RAM-i puhul kasutatakse mõlema pordi jaoks kõiki eelnevalt nimetatud signaale (eraldi ADDR-id, DIN-id, DOUT-id, WE-d ning CLK-id). *Simple Dual Port* RAM-is on aga üks port nähtud ette lugemiseks ja teine kirjutamiseks, mistõttu on selles elemendis üks DIN ja DOUT. Lugemis- ja kirjutamisaadressid on eraldi koos taktsignaalide ja vastavalt *Read Enable* ja *Write Enable* signaalidega, mis määravad, kas seda operatsiooni tehakse.

Samaaegsete operatsioonide tõttu võib ette tulla ka konflikte (loetavad andmed või mälupesal kirjutamisjärgselt asuvad andmed pole üheselt määratud) näiteks korraga samale pesale kirjutamisel või sama pesa lugemisel ja kirjutamisel. Kui samaaegselt kirjutada kahest pordist samale pesale erinevat infot, pole võimalik määrata pesa lõplikku seisu. Siiski sama pesa lugemisel ja kirjutamisel on kirjutamine alati edukas, kuid lugemise tulemus sõltub mälu seadistusest. [21]

Tabel 1. Mälupesa väärtus erinevatel mäluelemendi seadistuste korral. RF - *Read First* režiim, WF - *Write First* režiim, NC - *No Change* režiim, WE - *Write Enable*, DINA - Port A sisendvektor, DINB - Port B sisendvektor, DOUT - pordi väljundvektor, X - määramatu väärtus.

Portide takt-signaalid	**Port A kirjutamis-režiim**	**Port B kirjutamis-režiim**	**Port A WE signaal**	**Port B WE signaal**	**Port A DOUT signaal**	**Port B DOUT signaal**	**Lõplik mälupesa sisu**
Ühine	RF/WF/NC	RF/WF/NC	0	0	Varasem mälupesa sisu	Varasem mälupesa sisu	Varasem mälupesa sisu
Ühine	RF	RF/WF/NC	1 (DINA)	0	Varasem mälupesa sisu	Varasem mälupesa sisu	DINA
Ühine	WF	RF/WF/NC	1 (DINA)	0	DIA	X	DINA
Ühine	NC	RF/WF/NC	1 (DINA)	0	Varasem DOUT vektori sisu	X	DINA
Ühine	RF/WF/NC	RF	0	1 (DINB)	Varasem mälupesa sisu	Varasem mälupesa sisu	DINB
Ühine	RF/WF/NC	WF	0	1 (DINB)	X	DIB	DINB

Tabel 1. Mälupesa väärtus erinevatel mäluelemendi seadistuste korral. RF - *Read First* režiim, WF - *Write First* režiim, NC - *No Change* režiim, WE - *Write Enable*, DINA - Port A sisendvektor, DINB - Port B sisendvektor, DOUT - pordi väljundvektor, X - määramatu väärtus.

Portide takt-signaalid	**Port A kirjutamis-režiim**	**Port B kirjutamis-režiim**	**Port A WE signaal**	**Port B WE signaal**	**Port A DOUT signaal**	**Port B DOUT signaal**	**Lõplik mälupesa sisu**
Ühine	RF/WF/NC	NC	0	1 (DINB)	X	Varasem DOUT vektori sisu	DINB
Ühine	RF/WF/NC	RF/WF/NC	1	1	X	X	X
Erinevad	RF/WF/NC	RF/WF/NC	0	0	Varasem mälupesa sisu	Varasem mälupesa sisu	Varasem mälupesa sisu
Erinevad	RF	RF/WF/NC	1 (DINA)	0	Varasem mälupesa sisu	X	DINA
Erinevad	WF	RF/WF/NC	1 (DINA)	0	DINA	X	DINA
Erinevad	NC	RF/WF/NC	1 (DINA)	0	Varasem DOUT vektori sisu	X	DINA
Erinevad	RF/WF/NC	RF	0	1 (DINB)	X	Varasem mälupesa sisu	DINB
Erinevad	RF/WF/NC	WF	0	1 (DINB)	X	DINB	DINB
Erinevad	RF/WF/NC	NC	0	1 (DINB)	X	Varasem DOUT vektori sisu	DINB
Erinevad	RF/WF/NC	RF/WF/NC	1	1	X	X	X

Lisaks RAM-idele saab luua ka sarnased ROM-id, mis töötavad täpselt samamoodi, kuid ei kasuta DIN ja WE signaale, kuna sellele mäluelemendile ei saa andmeid juurde kirjutada, s.t saab ainult lugeda [21].

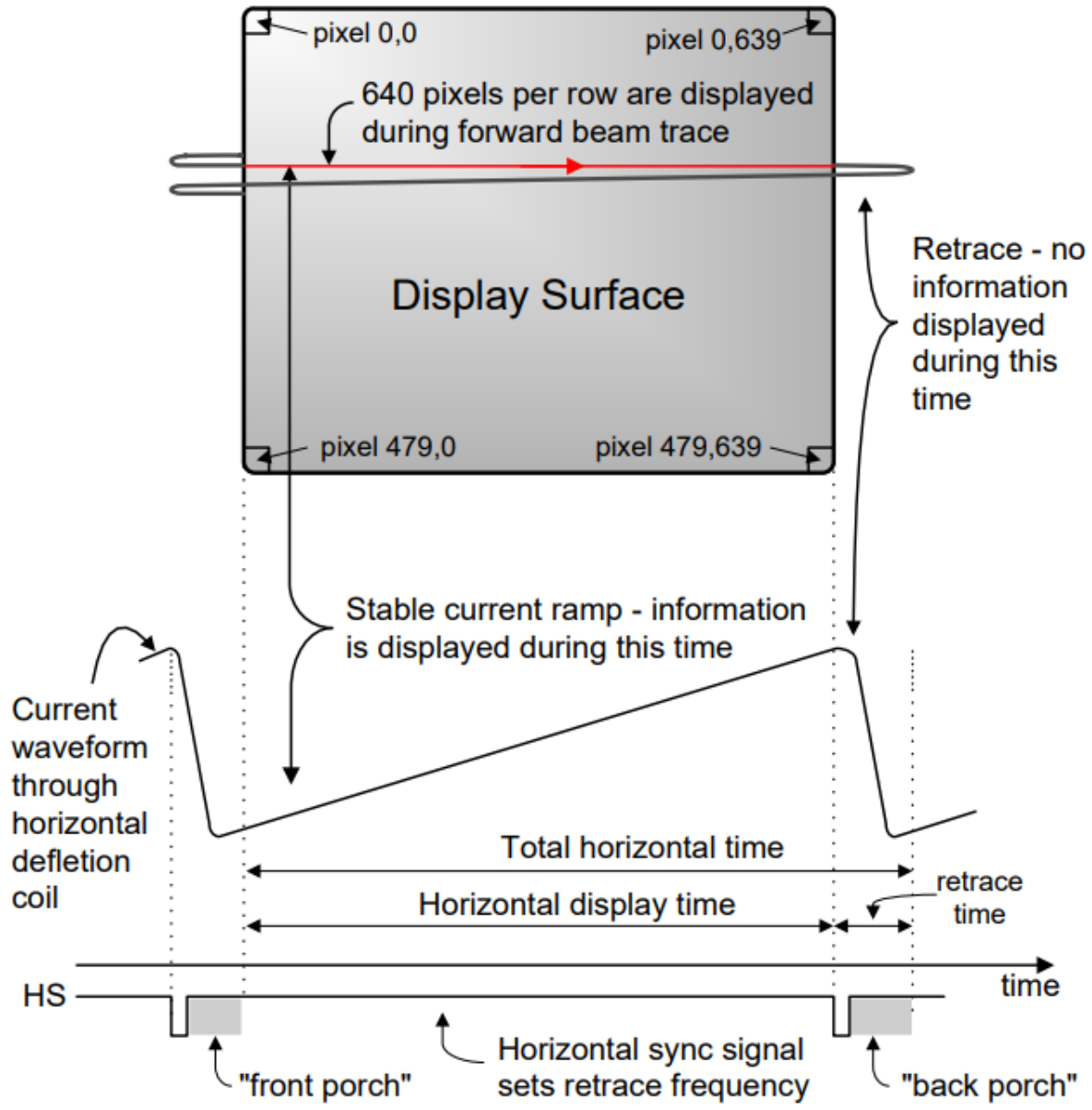
Mälukomponendi saab Block Memory Generatoris algväärtustada .coe faili abil, mis on Vivados kasutatav tekstifail, mille abil on võimalik IP Core'e luua. .coe faili esimesel real on kirjas, millises arvusüsteemis on järgneva ühemõõtmelise massiivi elemendid antud, ja järgmisel real defineeritakse massiiv, mille elemendid on eraldatud teineteisest komadega ja mis asuvad igaüks eraldi aadressil. [23]

3.4 VGA

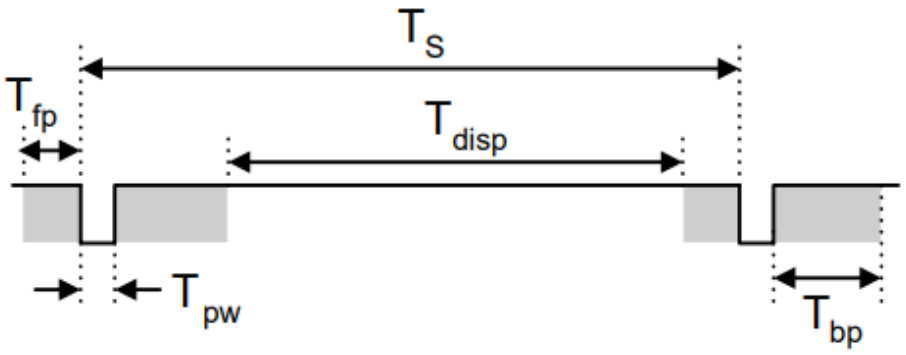
VGA on graafikastandard, mille abil on võimalik edastada ekraanile pilti [24]. Kuna algselt kasutati VGA standardit, elektronkiiretoru abil kuvatavatel piltidel, oli vaja täpselt ajastada kiire liikumine (Joonis 4). Selleks, et signaale omavahel sünkroonis hoida ja teada, millal kiir on liikumas tagasi algpositsioonile, tuleb edasi anda, millal on kiir liikumas paremalt vasakule ja alt üles. Seega on standardi järgi lisaks piksli värvi väärtustele edastada ka sünkroniseerimise signaalid, mis aitavad ka paika panna monitori eraldusvõime (Joonised 5 ja 6 ning Tabel 2). [12] [25] Basys3-e kasutusjuhendis määratud signaalide ajastamise tabeli kohaselt tuleb kasutada VGA pildi edastamiseks 25 MHz taktsignaali.

Basys3 arendusplaadilt monitorile pildi edastamiseks, on tarvis 14 signaali:

- punase tooni tugevus – 4-bitine signaal, mille iga bitt läheb erinevasse sisendisse;
- rohelise tooni tugevus – 4-bitine signaal, mille iga bitt läheb erinevasse sisendisse;
- sinise tooni tugevus – 4-bitine signaal, mille iga bitt läheb erinevasse sisendisse;
- horisontaalne sünkroniseerimise signaal;
- vertikaalne sünkroniseerimise signaal [12].



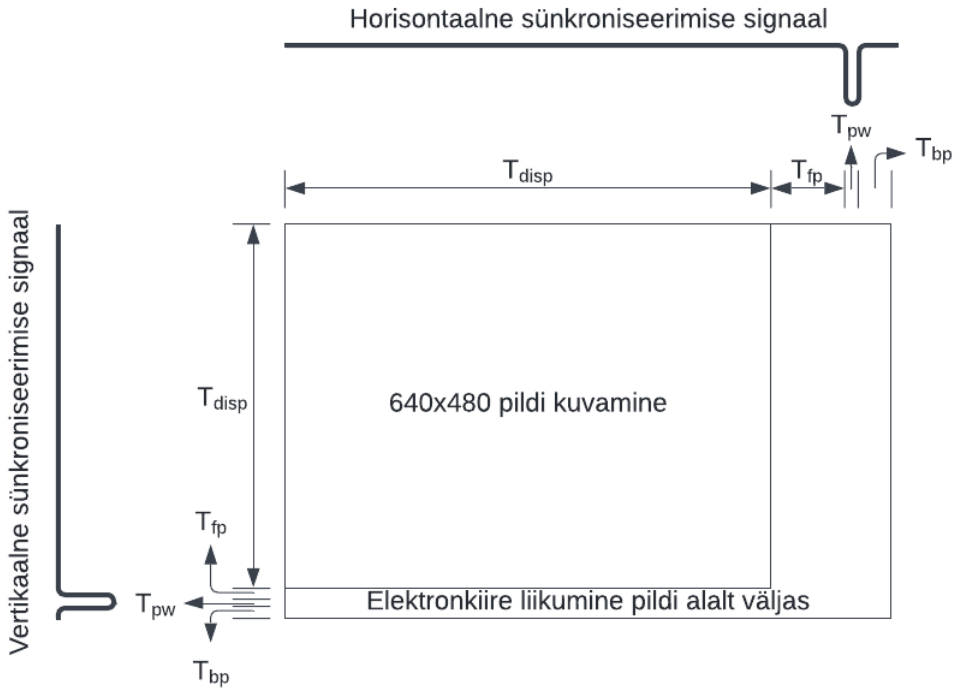
Joonis 4. Illustreeriv joonis elektronkiire liikumisest ekraanil. Lisatud on ka horisontaalse sünkroniseerimise signaal.



Joonis 5. Sünkroniseerimise signaalide muutumine vastavalt „elektronkiire“ liikumisele.

Tabel 2. „Elektronkiire“ liikumist kirjeldavad väärtused. Tabel näitab, kui kiiresti liigub kiir erinevates ekraani alades.

Tähistus Joonisel 5	Väärtus	Vertikaalne sünkroniseerimine			Horisontaalne sünkroniseerimine	
		Aeg	Takte	Ridu	Aeg	Takte
T_s	Täisring ümber terve ekraani (*Sync Pulse*)	16,7 ms	416800	521	32 µs	800
T_{disp}	Pildi osa ekraanist (*Display time*)	15,36 ms	384000	480	25,6 µs	640
T_{pw}	Sünkroniseerimise impulss (*Pulse width*)	64 µs	1600	2	3,84 µs	96
T_{fp}	Pildi järgne osa ekraanist (*Front porch*)	320 µs	8000	10	640 ns	16
T_{bp}	Pildile eelnev osa ekraanist (*Back porch*)	928 µs	23200	29	1,92 µs	48



Joonis 6. Horisontaalse ja vertikaalse sünkroniseerimise signaalide muutmine sõltuvalt „elektronkiire“ liikumisest.

4 Tulemused

4.1 Pildi kirjutamine mällu

Praktilise töö aluseks võttis autor pildi mõõtmetega 256x256 pikslit, mille töötlemiseks oli vaja see FPGA-le edastada. Alguses kasutas ta programmis .txt-laiendiga tekstifailist pildi pikslite väärtuste sisselugemiseks TextIO teeki, mille eesmärgiks on FPGA-väliste failide lugemine ja kirjutamine [26]. Siiski oli FPGA-välise tekstifaili rida-realt lugemine liiga ajakulukas ning nõudis liigselt operatsioone. Failist ühe rea sisse lugemiseks oli tarvis iga tähemärk eraldi sisse lugeda ja lõpuks need omavahel konkateneerida. Pärast neid operatsioone tuli terve sõne ümber täisarvuks teisendada ja korrata protsessi kõikide pildi pikslitega. Seetõttu oli mõistlikum kirjutada töödeldav pilt BRAM-i, millest oli võimalik ühe taktsignaali takti jooksul võimalik lugeda välja terve piksel õiges formaadis.

Enne mäluelemendi loomist kirjutas töö autor valmis töödeldava pildi .coe faili. Kuna töödeldav pilt oli värviline ja VGA adapter edastas igat värvitooni (punast, rohelist ja sinist) 4-bitiste vektoritena, kirjutas ta .coe faili igale reale ühe piksli värvuse 12-bitise binaararvuna, mille 4 vasakpoolset bitti edastasid punast, keskmised 4 bitti rohelist ja 4 parempoolset bitti sinist värvitooni (Joonis 7). Pärast .coe faili kirjutamist, alustas autor Block Memory Generatori abil *Single Port* RAM-i loomist (*Dual-Port*'i polnud tarvis, kuna korraga oli vaja lugeda ainult ühelt pesalt). *Write Width*'iks, mis määrab, kui pikad on igal mäluaadressil asuvad vektorid, kirjutas ta 12, kuna iga piksel kirjutatakse 12-bitistena. *Write Depth*'iks, mis määrab, mitu rida failis kokku on, kirjutas ta 65536 ehk 256x256 pildi pikslite arvu. Tähtis oli ka valida korrektne *Enable Port Type*'i väärtus. Kuna kõikidel mäluelementidel on ka olemas pordi töötamise signaal, mis lubab üldise mäluelemendi töötamise, oli lihtsam väärtustada *Enable Port Type* väärtusega *Always Enabled*. Vastasel juhul tulnuks luua tarkvaraliselt uus signaal, mis lubaks mäluelemendi kasutamise, ja see vastavale komponendile sisendiks anda. Viimase sammuna andis autor Block Memory Generatoris ette eelnevalt loodud .coe faili, mille järel oli võimalik loodud *Single Port* BRAM-ist pildi pikslite väärtusi lugeda (Lisa 2).

```
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
011101111101,
011101111101,
011101111101,
011110001101,
```

Joonis 7. Näide kasutatud .coe faili esimesest 6 reast. Esimesel real on märgitud, millises arvusüsteemis on faili sisu, teisel real alustatakse mäluaadresside täitmist ja seejärel kirjutatakse mäluaadressidele (alates 0-st) vastavad väärtused. Tähistatud on ka, millised bitid millist värvitooni kujutavad.

4.2 Pildi lugemine mälust VGA väljundisse

Selleks, et koodis kasutada mäluelementi kirjutatud andmeid, oli tarvis komponendile ette anda taktsignaal, aadress, millelt lugeda andmeid, ja 12-bitine vektor, kuhu kirjutada loetud andmed. Lisaks mäluelemendile tuli ka VGA-liidesele ette anda korrektsed signaalid. Kuna Basys3 arendusplaadil oli taktsignaali generaator sagedusega 100 MHz, siis muutis töö autor selle neli korda aeglasemaks ja sai soovitud 25 MHz signaali VGA jaoks. Õigele pikslile õige väärtuse andmiseks lõi ta kaks täisarvulist muutujat, mis kujutasid vastavalt piksli x- ja y-koordinaati. Autor kirjutas ka kaks paralleelset protsessi, mis neid muutujaid 25 MHz taktsignaali tõusva frondi järgi väärtustasid – kui x-koordinaadi väärtus jõudis 799-ni ehk „elektronkiire liikumise“ paremasse serva (Joonised 4 ja 6), kirjutas ta x-koordinaadi 0-ks, vastasel juhul suurendas ta x-koordinaati ühe võrra. Sarnaselt muutis ta y-koordinaadi väärtust – kui x-koordinaat jõudis 799-ni ja y-koordinaat 520-ni ehk alumine parem piksel, kirjutas ta y-koordinaadi 0-ks. Kui x-koordinaat jõudis üldiselt paremasse serva (y-koordinaat < 521), suurendas autor y-koordinaati 1 võrra. Peale „kiiritatava“ piksli muutmist, tuli muuta ka sünkroniseerimise signaale. Kui x-koordinaat oli läbinud kuvatava ekraani ulatuse (640 pikslit) ja *front porch*'i (16 pikslit) (Tabel 2) ehk oli jõudnud sünkroniseerimise alguspunkti (655. piksel), väärtustas autor horisontaalse sünkroniseerimise signaali 0-ks. Pärast seda, kui x-koordinaat oli ka üle sünkroniseerimise lõpp-punkti (kestis 96 pikslit ehk jõudnud 751. pikslini), kirjutas ta horisontaalse sünkroniseerimise signaali 1-ks. Sarnaselt toimis ta vertikaalse sünkroniseerimise signaaliga, mis oli väärtusega 0, kui y-koordinaadi väärtus oli vahemikus 489-491.

Kuna esimese piksli värvitoon on kirjutatud aadressile 0 ning iga järgnev ühe võrra suuremale aadressile, siis algväärtustas töö autor aadressi muutuja 0-ga, mida ta hakkas iga kord suurendama

ühe võrra 25 MHz taktsignaali tõusval frondil, kui x-koordinaat oli väiksem 640-st ja y-koordinaat väiksem 480-st. Kui x-koordinaat oli võrdne 640-ga ja y-koordinaat võrdne 479-ga, kirjutas ta aadressi uuesti 0-ks, kuna kogu pilt oli kuvatud ja alustades uuesti positsioonilt (0; 0) saab aadressi edaspidi taas ühe võrra suurendama hakata. Sellise lõpmatu tsükliga on võimalik kuvada mällu kirjutatud pilti ekraanile läbi VGA-liidese.

4.3 Pildi töötlemine

Mälusse kirjutatud pilti töötles töö autor erinevatel viisidel. Esmalt programmeeris ta juurde võimaluse muuta pilti heledamaks ja tumedamaks liugnuppude abil. Kuna kõik VGA-liidesesse edastatavad värvitoonid olid 4-bitised, kasutas ta pildi heledamaks muutmiseks samuti nelja liugnuppu, millest igaüks kujutas ühte bitti, ehk moodustus 4-bitine binaararv, mille võrra sai heledust muuta. Heleduse muutmiseks pidi autor kõiki värvitoone täpselt sama arvu võrra suurendama või vähendama. Kui muuta erinevaid toone erineva arvu võrra, muutub kolme põhitooni omavaheline suhe (kui suurendada ainult sinise ja rohelise tooni osakaalu, jääb pikslisse alles vähem punast tooni). Näiteks kui mälust loeti 12-bitine piksel väärtusega 0000 1000 1011 ehk RGB värvusruumis (0; 8; 11), millele lisati liugnuppudelt binaararvuline väärtus 0101 ehk kümnendkoodis 5, siis oli lõplik tulemus 0101 1101 1111 ehk (5; 13; 15). Ekraanil sellise tulemuse saavutamiseks luges töö autor mälust originaalse pildi piksli väärtuse, lisas sellele juurde loetud liugnuppude väärtuse ja kirjutas nende summa VGA-liidesesse. Siinkohal oli tähtis jälgida, et 4-bitiste värvitoonide summa ei oleks suurem kui 15, sest sellisel juhul tekiks ületäitumine ja piksel ei oleks kõikidel juhtudel heledam, vaid mõnikord ka tumedam. Ületäitumise korral suurendatakse (nt liidetakse või korrutatakse) bitivektorit mingi arvu võrra, mistõttu ei mahu vastus enam bitivektorisse ära (vastus on pikem kui vektori ruumi ette on antud). Sel juhul jätkatakse bitivektori väärtuse suurendamist otsast peale (alates 0 väärtusest), mille tulemusena pole liitmise vastus alati korrektne. Näiteks oleks eelmises näites võinud sinise tooni tugevus olla 15 asemel hoopis 16, mis 4-bitise binaararvu korral oleks hoopis väärtusega 0 ehk must. Sarnaselt lisas autor juurde ka pildi tumedamaks tegemise võimaluse, kus tuli jälgida, et liugnuppude väärtus ei muudaks pildi pikslist lahutamisel uut piksli väärtust negatiivseks. Eelneva näite korral: kui muuta punast tooni (väärtusega 0) 5 võrra tumedamaks saaksime -5, mis binaararvuna alatäitumist kontrollimata oleks olnud 1011 ehk kümnendarvuna 11.

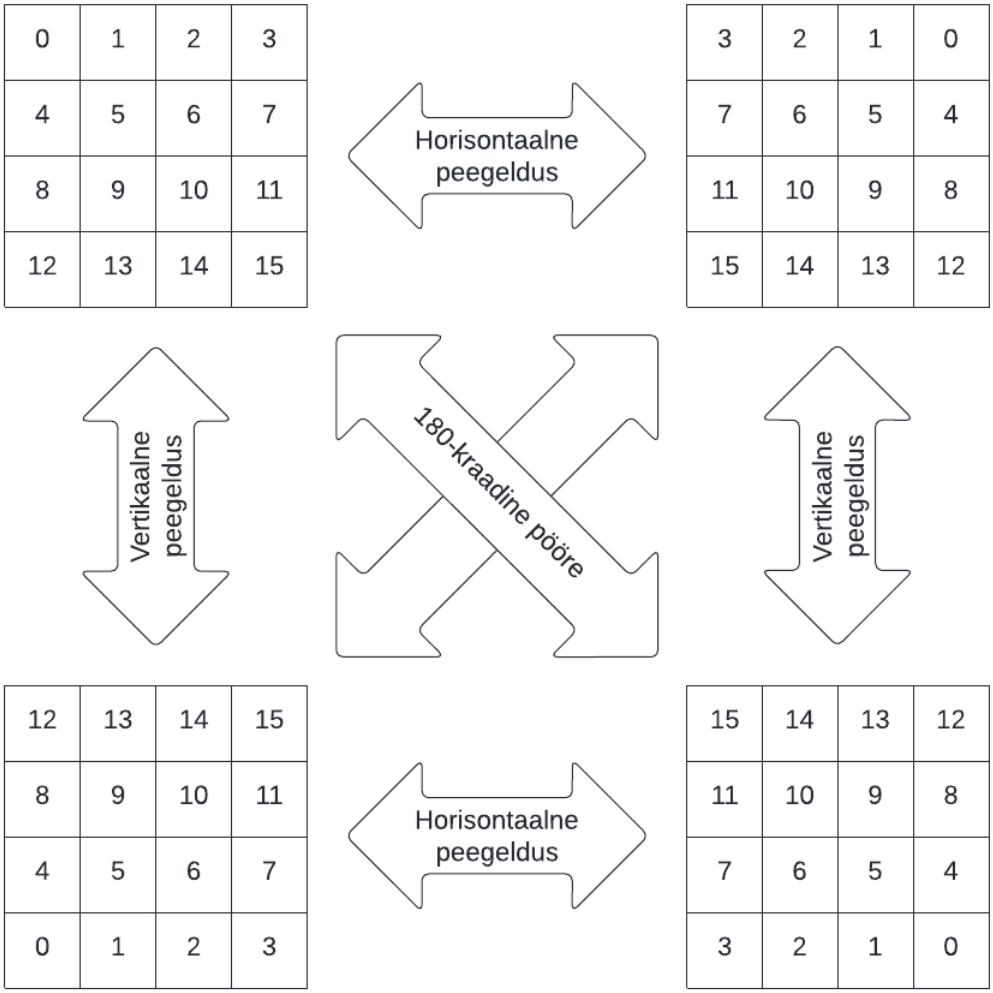
Teise töötluse võimalusena kirjutas töö autor juurde valiku peegeldada pilti nii horisontaalselt kui vertikaalselt. Kuna selles ülesandes polnud otseselt tarvis pikslite väärtusi muuta, siis sai

peegeldamise võimaluse paralleelselt juurde lisada. Selleks, et pilti horisontaalses suunas peegeldada, tuli muuta, millistelt mäluaadressidelt piksleid lugeda: ridade järjekord jääb samaks, kuid algse pildi esimene veerg on peegeldatud pildi viimaseks veeruks. Kui kirjutada pilti VGA-liidesesse vasakult paremale ja ülevalt alla, siis oli peegeldatud pildi esimeseks piksliks originaalse pildi esimese rea viimane piksel (mäluaadressil, mille väärtuseks oli ühe võrra väiksem arv kui pildi laius). Seejärel tuli igat järgnevat pikslit lugeda ühe võrra väiksemalt mäluaadressilt kuni rea lõpuni ehk 0 aadressini. Järgmiseks luges autor peegeldatud pildi teise rea algusesse originaalse pildi teise rea viimase piksli (mäluaadressilt, mille väärtuse ta sai, kui lisas eelnevale mäluaadressi väärtusele ühe võrra väiksema arvu kui kahekordse pildi laiuse), millest edasi tuli taas hakata mäluaadresse iga järgneva piksliga ühe võrra vähendama. Sarnast mustrit jätkates sai autor kuvada ekraanile horisontaalses suunas peegeldatud pildi (Joonis 8).

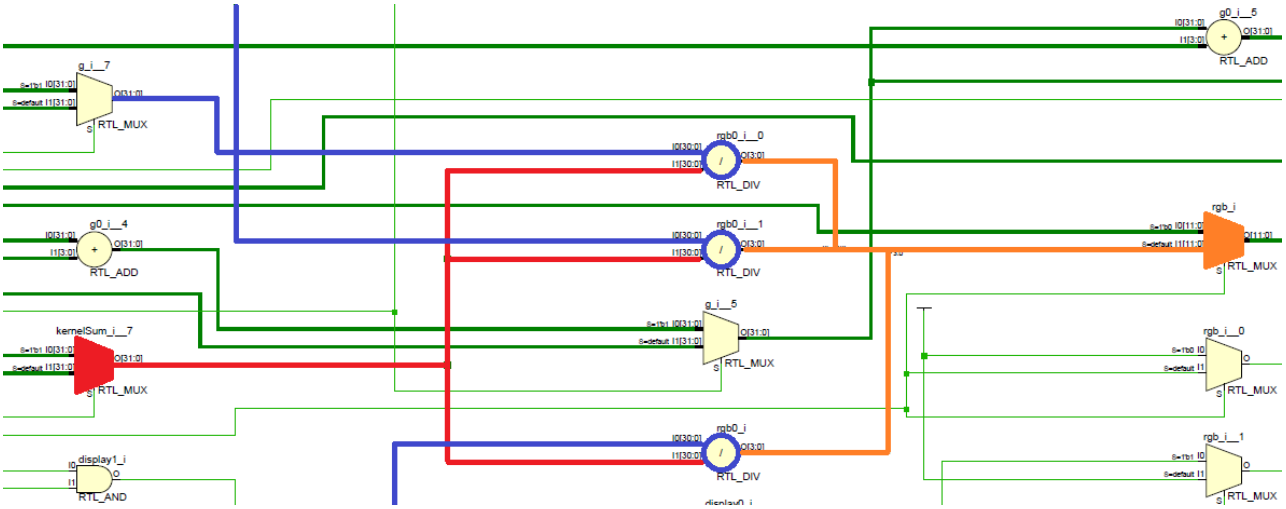
Vertikaalses suunas peegeldatud pildi puhul on veergude järjekord sama, kuid algse pildi esimene rida on peegeldatud pildi viimaseks reaks. Vertikaalse peegelduse sai töö autor, kui alustas lugemist mäluaadressilt: pildi laiuse ja kõrguse korrutise vahe pildi laiusega. Edasi tuli kuni rea lõpuni suurendada mäluaadressi ühe võrra. Järgmise rea esimese piksli mäluaadressi sai ta, kui lahutas eelnevast mäluaadressi väärtusest kahekordse pildi laiuse ja suurendas seda ühe võrra. Eelnevaid arvutusi korrates sai autor ekraanile lõpuks vertikaalselt peegeldatud pildi (Joonis 8).
Lõpuks kirjutas autor valmis ka koodi, mis kombineerib mõlemad peegeldused kokku ehk alguses peegeldab ühes teljes ning seejärel teises teljes ehk pöörab originaalset pilti 180° - nii read kui veerud on vahetuses. Peegeldatud pildi esimese rea esimeseks piksliks on viimase rea viimane piksel ehk mäluaadressil, mille ta sai kui vähendas pildi laiuse ja kõrguse korrutist ühe võrra. Edasi tuli vähendada igal lugemisel aadressi ühe võrra kuni pildi lõpuni välja (Joonis 8 ja Lisa 3).

Viimaseks töötluse meetodiks võttis töö autor hägustamise ehk kerneliga teostatava töötluse. Üle pildi libistatava „akna“ kasutamiseks oli kõige lihtsam kasutada kahekordset tsüklit. Tsükli lihtsamaks kasutamiseks lõi ta eraldi pilti kujutava (ehk samade mõõtmetega, kus pikslid olid vastavuses elementidega) maatriksi, kuhu lasi programmil esmakordsel käivitamisel lugeda pildi pikslite väärtused. Udustamise operatsiooni tegemiseks kasutas ta 3x3 kernelit, mistõttu lõi ta kaks üksteise sees olevat for-tsüklit, mis täitsid enda sisu 3 korda - nii käidi läbi kõik 9 „aknasse“ jäävat pikslit. Iga vaadeldava piksli puhul jagas autor piksli 12-bitise vektori 3-ks, mida oli võimalik töödelda paralleelselt - ta võttis värvitooni väärtuse, lisas aknas olevate teiste pikslite sama tooni summasse ja leidis selle summa jagatise „aknas“ olevate pikslite arvuga. Pärast vaadeldava piksli kõigi 3 värvitooni jagatise leidmist, sai autor VGA-liidesesse kirjutada 4-bitiste värvitoonide

konkatenatsiooni. Kuvades iga piksli väärtuseks teda ümbritsevate pikslite väärtuste keskmise, väljastas ta ekraanile hägustatud pildi (Joonis 9 ja Lisa 4).



Joonis 8. Maatriksi elementide (või ka pildi pikslite) peegeldused telgede suhtes.



Joonis 9. Hägustamise operatsiooni teostamine riistvaraliselt. Punasest trapetsist (multiplekser) liigub sinistesse ringidesse (iga värvitooni jagamistehe paralleelselt) arv, mis näitab, mitme piksli keskmist väärtust leitakse ehk mis oli „akna“ suurus. Sinistesse ringidesse liigub mööda siniseid jooni kõikide vaadeldud pikslite summa. Lõpuks konkateneeritakse 4-bitised (punane, roheline ja sinine) värvitoonid kokku üheks 12-bitiseks piksli väärtuseks oranžis trapetsis.

4.4 Praktikumijuhendid

Tudengitele VGA-liidese abil pildi ekraanile kuvamise tutvustamiseks jagas töö autor praktikumi kaheks osaks. Esimeseks osaks oli VGA-liidese tööle saamine, kus terve ekraan värvitakse 12 liugnupu abil kindlat tooni. Teiseks osaks on sama koodi kasutamine, kuid sinna juurde lisandus pildi kirjutamine *Single Port* RAM-i Block Memory Generatori abil ning seejärel mälust loetud info väljastamine VGA-liidesesse. Praktikumiülesannete eesmärgiks oli luua koodid, mille autor ise varasemalt kirjutas ja mis andsid tudengile visuaalselt kiiret tagasisidet lahendatud ülesande kohta. Juhendi kirjutamiseks võttis autor ette varasemalt kirjutatud koodi, jagas selle loogilistesse plokkidesse ning lisas juhendisse vastavad loogilised peatükid, mis võiksid luua parema arusaama läbitavast teooriast (Lisad 1 ja 2).

Pärast juhendite valmimist jagas töö autor neid sama õppeaine teiste praktikumide juhendajatega ning küsis tagasisidet. Saadud kommentaaride põhjal täiendas ta juhendeid - täpsustas selgitavat sisu, jooniseid, korrigeeris lausete sõnastusi. Kui juhendid said valmis, töötasid tudengid praktikumides need läbi. Praktikumid läbiti suures osas iseseisva tööna – tudengitele tehti Moodle’i keskkonnas nähtavaks praktikumi juhend, mille nad läbi töötasid. Seejärel asusid tudengid juhendis olevaid ülesandeid läbima, mille juures oli neil võimalus küsida abi õpperuumis olevalt praktikumijuhendajalt. Juhendaja eesmärk oli suunata tudengit õiges suunas mõtlema ning aidata tal

jõuda lahenduseni otsest abi osutamata. Kui tudeng sai ülesanded valmis, andis ta sellest juhendajale märku, pärast mida vaatas juhendaja tudengi lahendust (nii koodi kui füüsilisel plaadil toimuvat), küsis vajadusel täpsustavaid küsimusi ning hindas tudengi arusaamist teemast.

4.4.1 Tagasiside

Lõpuks küsis töö autor tudengitelt ka tagasisidet [27] loodud praktikumide kohta. Õppeainele oli kokku registreeritud 35 tudengit, kellest 23 lõpetasid edukalt praktikumid. Tudengitest, kes läbisid praktikumid, vastas küsimustikule 12. Tagasisidest selgus, et 10 tudengi jaoks 12-st oli praktikumide teooriaosa ning ülesande eesmärk arusaadavad. 9 tudengit väitsid, et said VGA ja BRAM-i õpetavate praktikumide raames abi juhendaja(te)lt, 2 tudengit ei osanud vastata ning 1 tudeng pigem ei saanud abi. 4 vastajat arvas, et praktikumid polnud liiga rasked, 5 jäi neutraalseks ja 3 leidis, et praktikumid olid pigem üle nende võimete. Vaatamata ülesannete raskusastmele leidis 11 tudengit 12-st, et ka järgmisel aastal võiks kasutada töö autori loodud juhendeid ning et nende jaoks olid praktikumid väärtuslikud.

Valminud praktikumide kõige keerulistemate osadena toodi välja taktsignaali loomist, piksli positsiooni väärtustamist ja arvutamist, sünkroniseerimise signaalide loomist ning koodi üksluisust ja kapriissust. Kui VGA käivitamiseks oli tarvis 25 MHz signaali, siis loodi algselt kogemata kaks korda suurema või väiksema sagedusega taktsignaale. Pildi pikslite väärtustamise juures tuli hoolega jälgida, millise piksli väärtuseni lubada pilti kuvada, või hoida sünkroniseerimise signaali madalana. Juba ühepiksline erinevus standardist lõi olukorra, kus ekraanile pilti ei kuvatud. Seejuures tuli ka vaadata, kas tingimuslausete operatsioone teostati õigete operaatoritega (näiteks operaatorid „<“ ja „≤“ käituvad erinevalt kahe võrdse väärtuse võrdlemisel). Samuti oli oluline jälgida, kas pildi pikslite nummerdamist alustati 1-st või 0-st.

Vabas vormis kommentaaride osas kiitsid tudengid juhendite autorit ning mainisid, et ülesanded olid toredad. Ühe tudengi puhul leidsid saadud teadmised kasutust igapäevases töökohas. Lisaks toodi välja, et koodist vigade otsimine on keeruline, kood ebaefektiivne ning VHDL on raske programmeerimiskeel.

5 Tulemuste analüüs ja järeldused

Tudengite antud tagasisidest võib eeldada, et töö autori loodud juhendid olid pigem edukad, kuna enamuse arvates olid need arusaadavad ja väärtuslikud. Siiski võiks enne nende kasutusele võtmist teha seal mõningaid muudatusi. Näiteks võiks üle vaadata kirjeldava teksti teooriaosa: leida rohkem paralleele reaalse eluga, lisada selgitavaid jooniseid või lisamaterjale näiteks õppevideo vormis. Selleks, et praktikumi ülesandeid lahendades kiiremini vigu leida, võiks soovitada juhendis simulatsiooni kasutamist. Simulatsioon loob lihtsama võimaluse lugeda, millise sagedusega on loodud taktsignaal, mis juhtub pildi pikslitega servades jms. Samuti võiks teha tihedamat koostööd praktikumijuhendajatega, kes peaksid oskama ise ülesandeid lahendada ning vajadusel abi osutada. Lisaks saab koodi efektiivsemaks muuta ning anda tudengitele vabamad käed koodi struktuuri osas (võib-olla oleks see nende jaoks arusaadavam). Viimase soovitusena võiks lasta tulevikus loodavad juhendid mõnel sihtgruppi kuuluval tudengil läbi lugeda, et enamik ülesannetest oleks arusaadavad ja edukalt tehtavad.

5.1 Edasised arendused

Et mitte piirduda ühe graafikastandardiga, võiks järgmisena implementeerida samad praktikumiülesanded näiteks HDMI-liidesel. See annab võimaluse õppida uut liidest, mis on tänapäevasem ja laialdasemalt kasutusel. Uue standardi õppimine aitab tudengitel analüüsida erinevaid graafikastandardeid ja kasutada saadud teadmisi vajadusel rohkem igapäevaelus.

6 Kokkuvõte

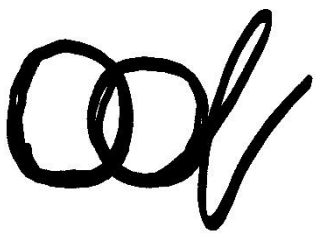
Praktikumijuhendite kirjutamiseks pidi töö autor iseseisvalt endale selgeks tegema erinevaid uued teemad, sh VGA standardi kasutamine, mäluelemendi loomine ning mõlema rakendamine programmeerimiskeeles VHDL. Lisaks autori oskustele täienevad ka tudengite teadmised ja oskused, kuna uued praktikumid rõhuvad funktsionaalsele lugemisele, kriitilisele mõtlemisele ning täpsusele. Õppeaine „Digitaalne loogika“ õpiväljundid saavad samuti paremini täidetud, kuna tundmatu graafika standardi kasutamine nõuab loogilist mõtlemist ja sarnaneb reaaleluliste ülesannetega.

Lisaks tudengite oskuste arendamisele sai töö autor ka rohkem ära kasutada arendusplaadi omadusi, sest nüüd on kasutusel ka plaadil olev VGA-liides. Pärast praktilise osa kirjutamist teab autor, et juhendite kirjutamisel on parem lisada rohkem lisamaterjale, joonised jms, sest nii tekib arusaam teemast kiiremini. Samuti teab autor, milliseid küsimusi võiks küsida praktikumi läbinud tudengitelt, et saada võimalikult abistavat tagasisidet ja leida murekohti (näiteks küsida keerulisemaid kohti koodi kirjutamisel, kas praktikum on üldjoontes vajalik jne). Lõpuks õppis autor ka leidma lahendusi uutele probleemidele – nii juhendis tekkinud arusaamatustele kui ka üldiselt VHDL eripäradele.

Tänuavaldused

Autor soovib tänada bakalaureusetöö juhendajat Margus Rosinat, kes leidis alati kõikidele muredele lahendusele, oskas iga kord soovitada mõnda head allikat või suunda, kuhu edasi liikuda.

Samuti on autor tänulik kaaspraktikumijuhendajatele, kes olid nõus uusi praktikume läbi viima ja aitama ainega seotud probleemidega, ning tudengeid, kes vastutulelikult läbisid praktikumid ja andsid tagasisidet.



Viited

[1] „Digitaalne loogika“, *Tartu Ülikooli Õppeinfosüsteem*. Vaadatud: 10.05.2022. [Võrgumaterjal] https://ois2.ut.ee/#/courses/LOTI.05.041/version/f3f2c2cd-812d-973d-340b-3cea9e45fd85/details.

[2] „About Us“, *Digilent*. Vaadatud: 10.05.2022. [Võrgumaterjal] https://digilent.com/shop/company/#about-digilent.

[3] R. Allik, „Pmod lisamoodulite kasutamine “Digitaalse loogika” aine praktikumides“, lk 55, Tartu, 2019.

[4] „Digitaalsüsteemid“, *TalTechi Õppeinfosüsteem*. Vaadatud: 16.05.2022. [Võrgumaterjal] http://ois.ttu.ee/aine/IAS0150.

[5] S. Jingcheng, W. Zhengyan, ja L. Zenggang, „Implementation of Sobel Edge Detection algorithm and VGA display based on FPGA“, 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), 2019, lk 2305–2310. DOI: 10.1109/IAEAC47372.2019.8997533.

[6] T. M. Khan, D. G. Bailey, M. A. U. Khan, ja Y. Kong, „Real-time edge detection and range finding using FPGAs“, *Optik*, kd 126, lk 1545–1550, 2015, DOI: 10.1016/j.ijleo.2015.01.024.

[7] „What is an FPGA? Field Programmable Gate Array“, *Xilinx, Inc*. Vaadatud: 23.10.2021. [Võrgumaterjal] https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[8] „7 Series FPGAs Data Sheet Overview“, *Xilinx, Inc*. 19 lk. Vaadatud: 23.10.2021. [PDF] https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[9] N. Dubey, „Top 5 FPGAs Manufacturers in the World“, *BISinfotech*. 17.09.2020. Vaadatud: 10.05.2022. [Võrgumaterjal] https://www.bisinfotech.com/top-5-fpgas-manufacturers-in-the-world/.

[10] „Company Overview“, *Xilinx, Inc*. Vaadatud: 23.10.2021. [Võrgumaterjal] https://www.xilinx.com/about/company-overview.html.

[11] „Vivado ML Overview“, *Xilinx, Inc*. Vaadatud: 10.05.2022. [Võrgumaterjal] https://www.xilinx.com/products/design-tools/vivado.html.

[12] „Basys3™ FPGA Board Reference Manual“, *Digilent, Inc*. 12.08.2014, lk 19. Vaadatud: 23.10.2021. [PDF] https://www.xilinx.com/content/dam/xilinx/support/documentation/university/XUP%20Boards/XUPBasys3/documentation/Basys3_rm_8_22_2014.pdf

[13] „Basys 3“, *Digilent, Inc*. Vaadatud: 28.04.2022. [Võrgumaterjal] https://digilent.com/reference/programmable-logic/basys-3/start.

[14] G. Anbarjafari, „Introduction to image processing“. Vaadatud: 11.05.2022. [Võrgumaterjal] https://sisu.ut.ee/imageprocessing/book/1

[15] J. Sachs, „Digital Image Basics“. lk 14. Vaadatud: 16.04.2022. [PDF] https://www.dl-c.com/Temp/downloads/Whitepapers/Basics.pdf.

[16] E.-M. Pulfer, „Different Approaches to Blurring Digital Images and Their Effect on Facial Detection“, lk 32, Fayetteville, 2019.

[17] „Intellectual Property“, *Xilinx, Inc*. Vaadatud: 28.04.2022. [Võrgumaterjal] https://www.xilinx.com/products/intellectual-property.html.

[18] „Binary Counter“, *Xilinx, Inc*. Vaadatud: 28.04.2022. [Võrgumaterjal] https://www.xilinx.com/products/intellectual-property/binary_counter.html.

[19] „DisplayPort Subsystem“, *Xilinx, Inc.* Vaadatud: 28.04.2022. [Võrgumaterjal] https://www.xilinx.com/products/intellectual-property/ef-di-displayport.html.

[20] „Block Memory Generator“, *Xilinx, Inc*. Vaadatud: 28.04.2022. [Võrgumaterjal] https://www.xilinx.com/products/intellectual-property/block_memory_generator.html.

[21] „7 Series FPGAs Memory Resources User Guide“, *Xilinx, Inc*. 03.07.2019, lk 88. Vaadatud: 08.04.2022. [PDF] https://docs.xilinx.com/api/khub/documents/9gZGbqBxtlKXxBfkBt~lAg/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=3.11.20&filename=ug473_7Series_Memory_Resources.pdf

[22] „Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS181)“, *Xilinx,* Inc. 21.12.2017, lk 63. Vaadatud: 11.05.2022. [PDF] https://www.marutsu.co.jp/contents/shop/marutsu/ds/ds181_Artix_7_Data_Sheet.pdf

[23] „Vivado Design Suite User: Guide Designing with IP“, *Xilinx,* Inc. 03.11.2021, lk 114. Vaadatud: 28.04.2022. [PDF] https://docs.xilinx.com/api/khub/maps/87gBuz1fSgU0_7SkMU7_lw/attachments/N7Owaddv80jD_RKUMlONmw/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=3.11.23&filename=ug896-vivado-ip.pdf

[24] T. Fisher, „What Does VGA Mean?“, *Lifewire*. 21.01.2022. Vaadatud: 10.04.2022. [Võrgumaterjal] https://www.lifewire.com/what-is-vga-2626027.

[25] „VGA Controller (VHDL)“, *Digi-Key Electronics*. 17.03.2021. Vaadatud: 10.04.2022. [Võrgumaterjal] https://forum.digikey.com/t/vga-controller-vhdl/12794.

[26] „TextIO“, *VHDL Guide*. 22.09.2017. Vaadatud: 28.04.2022. [Võrgumaterjal]

https://vhdlguide.com/2017/09/22/textio/.

[27] O. Oll, „Tagasiside VGA praktikumide kohta“, *Google Forms*. Vaadatud: 16.05.2022. [Võrgumaterjal] https://forms.gle/gzs8TGG7FfPWEyeDA.

Lisad

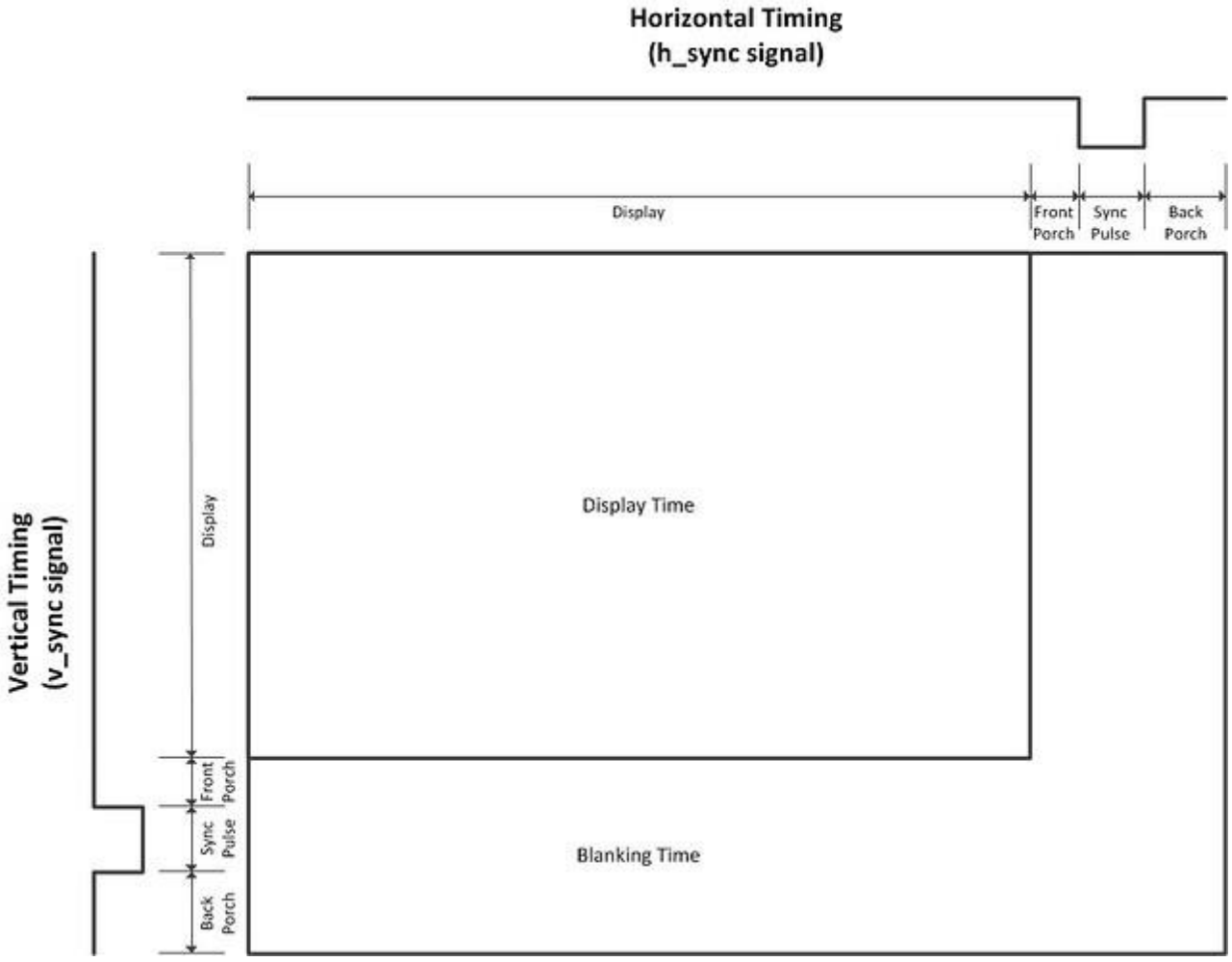
Lisa 1. Praktikumi VGA I juhend

Digitaalse loogika VI praktikum: VGA I

10 punkti

1. Sissejuhatus

VGA-ks (Video Graphics Array) nimetatakse nii riistvaralist graafika kontrollerit kui ka graafika standardit. 1987. aastal alguse saanud standardis sisalduvad lisaks kolme põhivärvi signaalile ka sünkroniseerimise signaalid, mida oli tarvis kasutada elektronkiiretoru abil kuvatavatele piltidele (Basys3 kasutusjuhend joonis 12). Kuna kiire liikumisel üle ekraani oli vajalik juurde arvestada kiire liikumise kiirus ning aeg algpositsioonile naasemiseks, pidid kiir ja värviinfo olema sünkroonis (Basys3 kasutusjuhend joonis 13). Tänapäeval kasutusel olevates vedelkristallkuvarites enam elektronkiirt ei kasutata, kuid VGA standardi kohased signaalide ajastused on jäänud samaks. Selle praktikumi eesmärgiks on aru saada, kuidas töötab VGA ning kuvada selle abil pilt ekraanile.



Joonis 1. Illustreeriv joonis VGA sünkroniseerimise signaalidest. [Digi-Key TechForum]

2. Konstandid

Loo Vivados uus projekt, kus hetkel võid jätta I/O pordid defineerimata (täpsed sisendid ja väljundid lisame *entity*'sse hiljem). Loo enda loodud arhitektuuri *constant*-tüüpi muutujad kuvatava pildi kõrguse ja laiuse jaoks. Lisa ka konstandid vertikaalse ning horisontaalse *front porch*, *back porch* ja *pulse width* (joonisel 1 kui *Sync Pulse*) väärtuste jaoks. Praktikumi raames loome VGA puhul tavapärase 640x480 pildi. Otsitavad väärtused leiad kasutusjuhendist. Nimeta konstandid nii, nagu need on kasutusjuhendis.

3. Kohandatud taktsignaal

Pildi edastamiseks ekraanile peavad meie signaalid liikuma kindla kiirusega. Selleks on vajalik luua eraldi taktsignaal. Leia kasutusjuhendist, millise sagedusega kella on meie VGA kasutamiseks vaja, loo arhitektuuri *signal*-tüüpi muutuja ning protsess, mis seda signaali vastavalt muudab.

4. Veel muutujaid

Samuti oleks mõistlik kasutada kahte täisarvulist muutujat, mis määravad, millist pikslit meie „elektronkiir" kiiritab. Lisa arhitektuuri vastavad *signal*-tüüpi muutujad. Soovituslik on need ka algväärtustada.

5. Entity

Lisame nüüd varasemalt tühjaks jäänud *entity*'sse sisendid ja väljundid. Täpsed väljundid (milliseid signaale VGA port pildi edastamiseks vajab) ja nende vektorite suurused leiad kasutusjuhendist (nimeta need signaalid samuti kasutusjuhendi järgi). Sisendiks on vaja taktsignaali ja liugnuppe vastavalt värvitoonide signaalide vektorite suurustele (praktikumi lõpus saame muuta reaalajas igat edastatavat värvi bitti).

6. Piksli asukoht

Meie „elektronkiir" on ka tarvis liikuma panna. Selleks kirjuta kaks protsessi, mis muudavad 4. punktis loodud pikslit vastavalt kahes teljes. Mõtle, milline signaal tuleks lisada *sensitivity list*'i (millistel hetkedel tuleb „elektronkiirt" liigutada) ja kuidas piksli väärtused muutuvad terve pildi kuvamise vältel (Basys3 kasutusjuhend joonis 13 ning käesoleva juhendi joonis 1).

7. Sünkroniseerimine

Selleks et meie värve ja piksleid omavahel sünkroonis hoida, tuleb VGA-le ette anda ka vertikaalse ja horisontaalse sünkroniseerimise signaalid. Loo veel kaks protsessi, mis muudavad eraldi mõlemat sünkroniseerimise signaali vastavalt VGA standardile (joonis 1).

8. Pildi väljastamine

Samuti on tarvis veel ka ette anda, milliseid toone igal pikslil kuvatakse. Loo protsess, mis kirjutab VGA värvitoonide sisenditesse vastavad liugnuppude väärtused (näiteks kui soovida punast ekraani, tuleb kõik punase värvi bittide liugnupud kõrgeks lükata ja teiste värvide liugnupud madalaks, kusjuures iga liugnupp määrab ühe biti). Juhul, kus „elektronkiirega" kiiritatav piksel pole ekraani (640x480) sees, tuleb kirjutada kõikide värvitoonide väärtuseks 0. Loodava koodiga saad luua ühevärvilist pilti, kuid ka ambitsioonikamad lahendused on teretulnud.

9. *Constraints* fail

Viimase sammuna tuleb projekti lisada *constraints* fail. Taaskord leiad abi kasutusjuhendist.

10. Näita töötav tulemus juhendajale ette

Oled praktikumi edukalt läbinud, kui liugnuppudega on võimalik värvida ekraan erinevatesse värvitoonidesse.

Lisa 2. Praktikumi VGA II juhend

Digitaalse loogika VII praktikum: VGA II

5 punkti

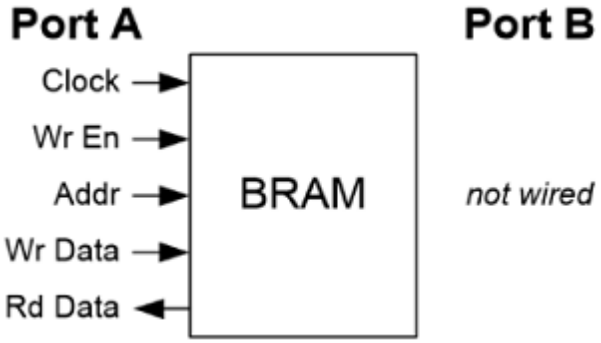
1. Sissejuhatus

VGA II praktikumi eesmärgiks on arendada edasi saadud teadmisi VGA-st ning mõista BRAM-i olemust ja kasutamist. Kui 4. praktikumis kasutasime *Distributed* RAM-i, siis selles praktikumis kasutame BRAM-i (*Block Random Access Memory*). Suurimaks erinevuseks on see, et *Distributed* RAM-i korral kirjutatakse andmed üle FPGA laiali LUT-idesse, kuid BRAM-i korral kirjutatakse need ühte kindlasse kohta, mis võimaldab mahutada rohkem andmeid kui *Distributed* RAM (lisalugemist).

Lisaks on Vivado tarkvaras võimalik graafiliselt luua erinevaid mooduleid IP (*Intellectual Property*) *Core*-ide abil. IP *Core* tähistab valmis kujul lihtsamat koodiplokki, mida saab kergesti komponendina projekti lisada ja seejärel kasutada (lisalugemist). Selles praktikumis kasutamegi *Block Memory Generator*'it, mille abil loome BRAM komponendi.

Vivado võimaldab luua nii *Single Port* RAM-i kui ka *Dual Port* RAM-i. *Single Port* RAM võimaldab ühel taktil kas ainult lugeda või ainult kirjutada kindlale mäluaadressile. BRAM kasutab tööks järgnevaid signaale (joonis 1):

- Clock - taktsignaal
- Wr En - määrab, kas loeme mälust või kirjutame mällu
- Addr - aadress, millelt loeme või millele kirjutame andmeid
- Wr Data - neid andmeid kirjutab eelnevalt määratud mäluaadressile
- Rd Data - sinna loeb andmeid eelnevalt määratud mäluaadressilt



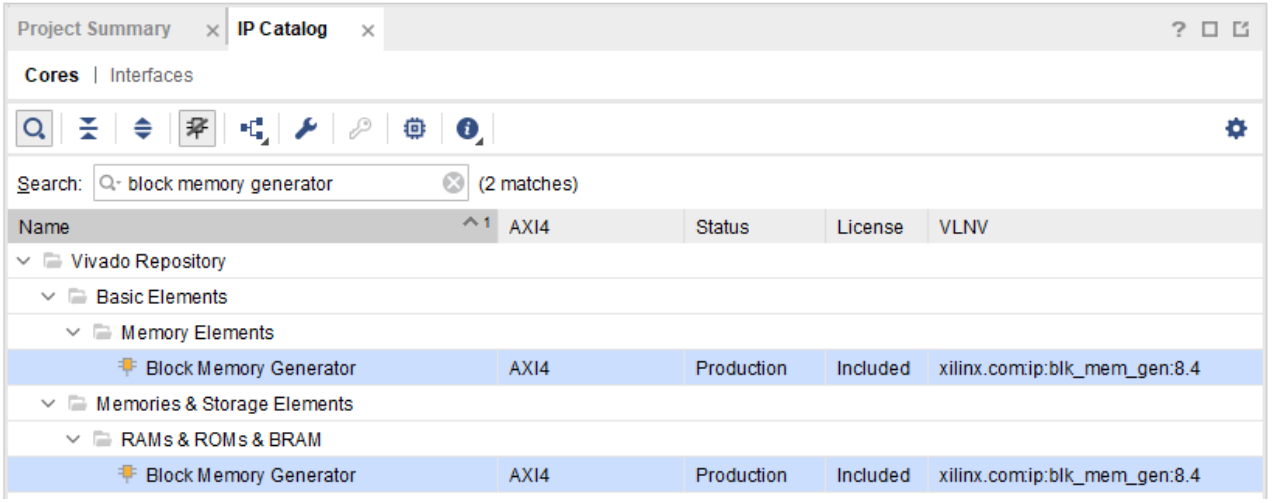
Joonis 1. *Single Port* BRAM-i illustreeriv joonis. [https://www.nandland.com/articles/block-ram-in-fpga.html]

Erinevalt Single Port RAM-ist omab *Dual Port* RAM kahte porti, mistõttu on võimalik korraga kuni kahe mälupesaga opereerida. Selles praktikumis kasutame siiski *Single Port* BRAM-i ja kirjutame ühe pildi FPGA mällu, kust me selle programmi käigus välja loeme ning ekraanile kuvame.

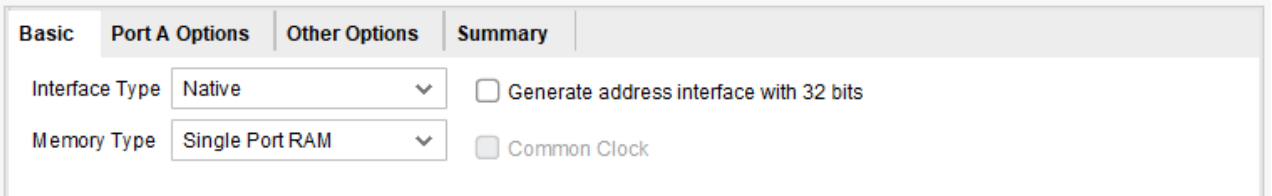
2. Pildi kirjutamine mällu

BRAM-i loomiseks on vaja .coe laiendiga faili, mida kasutatakse mäluga seotud IP *Core*'ide juures, et RAM-i või ROM-i andmeid salvestada. Kuna kirjutame mällu ühe 256x256 pildi ühedimensionaalse maatriksi kujul, siis on meie .coe failis igal real kirjeldatud ühe piksli värvitoonide väärtus. Lisaks värvitoonidele peab olema ka .coe faili alguses määratud, millises arvusüsteemis on andmed failis antud. Võid julgelt uurida, milline on praktikumi materjalidega kaasa antud .coe faili sisu.

Esimeseks ülesandeks on genereerida Vivados BRAM element, millesse kirjutame pildi. Ava vasakult menüüst „IP Catalog“, kirjuta avanenud aknas otsingusse „Block Memory Generator“ ja tee topeltklõps ühel vasteks tulnud objektil (need on samad generaatorid).

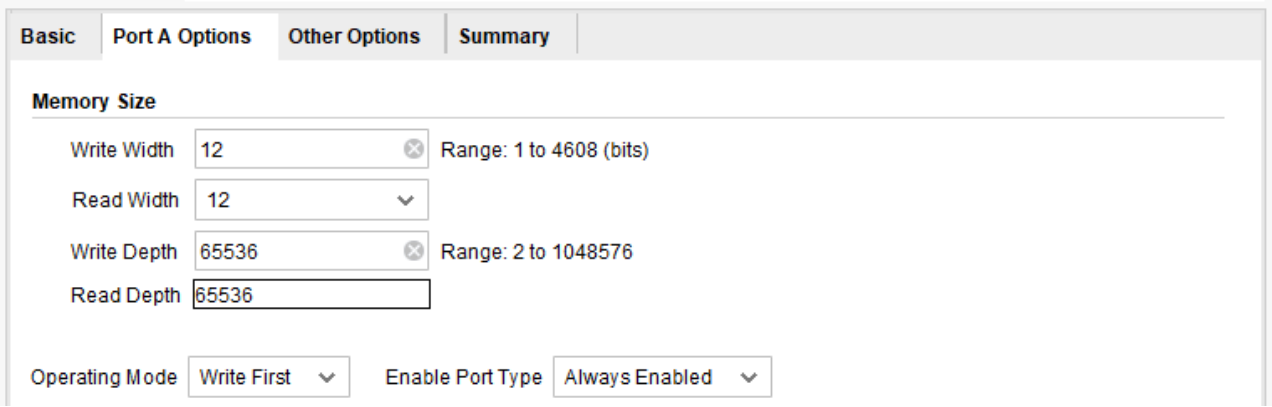


Avanenud generaatoris määra „Component Name“ lahtris enda BRAM-ile sobilik nimi ja vali „Memory Type“-iks „Single Port RAM“.



Praktikumide failide seas oleva coe-failis on defineeritud kõik pildi pikslid 12-bitiste vektoritena, mille 4 parempoolset bitti tähistavad punast värvitooni, 4 keskmist bitti rohelist värvitooni ja 4 vasakpoolset bitti sinist värvitooni.

„Component Name“ lahtri all olevalt ribalt vali sakk „Port A Options“. „Write Width“ väärtuseks määra 12, sest iga failis oleva rea pikkus on 12 tähemärki. Kuna pildi suurus on 256x256 pikslit, on failis kokku 256x256 rida ja „Write Depth“ väärtuseks 65536. „Enable Port Type“ väärtuseks määra „Always Enabled“.



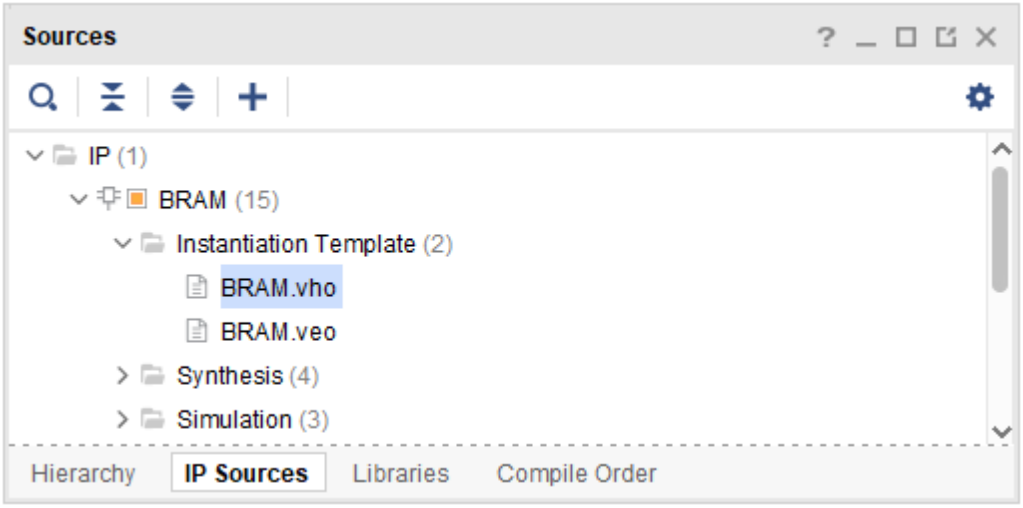
„Other Options“ saki all tee linnuke kasti „Load Init File“ ette ja selle all olevasse lahtrisse määra etteantud coe-fail.

„Summary“ saki alt näed infot loodud BRAM-i kohta. Jäta meelde, milline on viimasel real defineeritud aadressiriba laius bittides, ning mõtle, miks see on selline. Vajuta „OK“ ja järgmises aknas „Generate“. Kui Vivado on lõpetanud BRAM-i sünteesi, saame alustada VHDL-koodi kirjutamist.

3. Mälu kasutamine koodis

Kopeeri VGA esimese praktikumi kood oma uude vhd-faili - see on meile põhjaks, mida hakkame muutma.

Esmalt lisa juurde veel kaks arusaadavalt nimetatud konstanti, mis kujutavad pildi pikkus ja laiust.
Järgmiseks loo arhitektuuri sisse *component* enda BRAM-ist, mille portide loetelu leiad, kui valid „Sources“ akna alumisest servast „Hierarchy“ asemel „IP Sources“, avad enda BRAM komponendi alamkausta, „Instantation Template“ kausta ja vho-tüüpi faili. vho-faili lõpus on kirjas, milliste nimedega, muutuja tüüpidega ja vektori suurustega pordid on vaja defineerida vhd-failis BRAM-i komponendi loomiseks. Siit leiad ka ühe tuttava arvu - aadressiriba laiuse.



Nüüd tuleb BRAM-i kasutamiseks lisada arhitektuuri ka reaalselt koodis kasutatavad signaalid (nimeta need arusaadavuse eesmärgil samade nimedega, nagu on komponendi signaalid), mille saab luua *component*'i signaalide eeskujul. Ära unusta neid algväärtustada. Ühte neist portidest pole siiski vaja luua, sest see signaal on meil juba olemas. Milline?
Kui *component* on loodud, tuleb lisada ka *Port Map* oma *component*'ist, kus igale *component*'i pordile vastab eelnevalt loodud signaal. Heaks näiteks *Port Map*'ist on esimese praktikumi simulatsiooni fail, kus oli samuti tarvis luua *component* oma vhd-failist ja teha *Port Map*.

4. Pildi lugemine mälust

Kui eelmises praktikumis kirjutasime koodi viimases protsessis värvitoonid otse liugnuppude abil VGA väljudisse, siis nüüd oleks tarvis lugeda õiged väärtused mälust. Mälust lugemine toimub *component*'ile etteantud taktsignaali tõusval taktil. Lisaks taktile peab olema õigesti määratud *Write Enable* signaal ning sellega seonduvalt võime jätta ka tähelepanuta ühe BRAM pordi signaali. Seega tuleb meil tegeleda koodi viimases protsessis ainult kahe BRAM pordi signaaliga. Kirjuta vastav protsess nii ümber, et pildi esimene piksel loetakse mälu esimeselt aadressipesalt ja iga järgnev piksel järgmiselt pesalt. Jälgi ka seda, et sa faili lugemisel aadressiruumist välja ei lähe. Eriti kõrgelt on hinnatud lahendused, kus loetud pilt ei asu ekraani nurgas, vaid keskel.

5. *Constraints* fail

Viimase asjana oleks viisakas ka eelnevast praktikumist võetud xdc-faili kohendada.

6. Näita töötav tulemus juhendajale ette

Oled praktikumi edukalt läbinud, kui näed ekraanil Lenat.

Lisa 3. VHDL kood mälust loetud pildi pööramiseks ning heleduse muutmiseks

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

-- Kasutatud allikas: https://forum.digilentinc.com/topic/17598-display-image-using-vga-from-block-ram/
-- Kasutatud allikas: https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/

entity VGA is
    Port ( clk : in STD_LOGIC;
           hSync : out STD_LOGIC;
           vSync : out STD_LOGIC;
           sw : in STD_LOGIC_VECTOR (15 downto 0);
           rgb : out STD_LOGIC_VECTOR (11 downto 0));
end VGA;

architecture VGA_arch of VGA is

-- VGA
signal clk_div : STD_LOGIC := '0'; -- 25MHz kell
signal hPos : integer := 0; -- muudetava piksli koordinaadid
signal vPos : integer := 0;
signal display : STD_LOGIC := '0'; -- kas pilti edastada

constant hDisplay : integer := 640; -- ekraani laius
constant hLeftBorder : integer := 48; -- ekraani back porch
constant hRightBorder : integer := 16; -- ekraani front porch
constant hRetrace : integer := 96; -- mingi nihe

constant vDisplay : integer := 480; -- ekraani kõrgus
constant vBottomBorder : integer := 10; -- ekraani front porch
constant vTopBorder : integer := 29; -- ekraani back porch
constant vRetrace : integer := 2; -- mingi nihe

-- Pilt
constant hImage : integer := 256; -- pildi laius
constant vImage : integer := 256; -- pildi kõrgus
```

```
constant topCoordinate : integer := vDisplay / 2 - vImage / 2; -- pildi ülemise serva koordinaat
constant bottomCoordinate : integer := vDisplay / 2 + vImage / 2; -- pildi alumise serva koordinaat
constant leftCoordinate : integer := hDisplay / 2 - hImage / 2; -- pildi vasaku serva koordinaat
constant rightCoordinate : integer := hDisplay / 2 + hImage / 2; -- pildi parema serva koordinaat

-- BRAM
signal wea : STD_LOGIC_VECTOR(0 downto 0) := "0";
signal addra : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal dina: STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
signal douta : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');

component image_rom is
   PORT (
        clka : IN STD_LOGIC;
        wea : IN STD_LOGIC_VECTOR(0 downto 0);
        addra : IN STD_LOGIC_VECTOR(15 downto 0);
        dina : IN STD_LOGIC_VECTOR(11 downto 0);
        douta : OUT STD_LOGIC_VECTOR(11 downto 0));
end component;

begin

-- BRAM
U1: image_rom Port Map (clka=>clk_div, wea=>wea, addra=>addra, dina=>dina, douta=>douta);

-- 25 MHz kell
process(clk)
variable clk_counter : STD_LOGIC := '0';
begin
   if rising_edge(clk) then
      clk_counter := not clk_counter;
      if clk_counter = '0' then
         clk_div <= not clk_div;
      end if;
   end if;
end process;

-- Piksli asukoha horisontaalne muutmine
process(clk_div)
begin
   if rising_edge(clk_div) then
      if hPos = hDisplay + hLeftBorder + hRightBorder + hRetrace then
         hPos <= 0;
      else
         hPos <= hPos + 1;
      end if;
   end if;
end process;

-- Piksli asukoha vertikaalne muutmine
```

```
process(clk_div)
begin
   if rising_edge(clk_div) then
      if hPos = hDisplay + hLeftBorder + hRightBorder + hRetrace then
         if vPos = vDisplay + vTopBorder + vBottomBorder + vRetrace then
            vPos <= 0;
         else
            vPos <= vPos + 1;
         end if;
      end if;
   end if;
end process;

-- Horisontaalse sünkronisatsiooni signaal
process(clk_div)
begin
   if rising_edge(clk_div) then
      if (hPos < hLeftBorder + hDisplay + hRightBorder + hRetrace) then
         hSync <= '1';
      else
         hSync <= '0';
      end if;
   end if;
end process;

-- Vertikaalse sünkronisatsiooni signaal
process(clk_div)
begin
   if rising_edge(clk_div) then
      if (vPos < vTopBorder + vDisplay + vBottomBorder + vRetrace) then
         vSync <= '1';
      else
         vSync <= '0';
      end if;
   end if;
end process;

-- Kunas pilti edastada
process(hPos, vPos)
begin
   if (hPos <= rightCoordinate) AND (hPos > leftCoordinate) AND (vPos <= bottomCoordinate)
AND (vPos > topCoordinate) then
      display <= '1';
   else
      display <= '0';
   end if;
end process;

-- Edastatav pilt
process(clk_div)
```

```
begin
   if rising_edge(clk_div) then
      if (display = '1') AND (sw(15) = '1') then
         for i in 0 to 2 loop
            if TO_INTEGER(unsigned(douta(i*4+3 downto i*4))) + TO_INTEGER(unsigned(sw(3
downto 0))) < 15 then
               rgb(i*4+3 downto i*4) <= std_logic_vector(unsigned(douta(i*4+3 downto i*4)) +
unsigned(sw(3 downto 0)));
            else
               rgb(i*4+3 downto i*4) <= "1111";
            end if;
         end loop;
       elsif (display = '1') AND (sw(15) = '0') then
         for i in 0 to 2 loop
            if TO_INTEGER(unsigned(douta(i*4+3 downto i*4))) > TO_INTEGER(unsigned(sw(3
downto 0))) then
               rgb(i*4+3 downto i*4) <= std_logic_vector(unsigned(douta(i*4+3 downto i*4)) -
unsigned(sw(3 downto 0)));
            else
               rgb(i*4+3 downto i*4) <= "0000";
            end if;
         end loop;
      elsif display = '0' then
         rgb <= (others => '0');
      end if;

      if (sw(14) = '0') AND (sw(13) = '0') AND (hPos = rightCoordinate) AND (vPos =
bottomCoordinate) then
         addra <= (others => '0');
      elsif (sw(14) = '0') AND (sw(13) = '0') AND (display = '1') then
         addra <= std_logic_vector(unsigned(addra) + 1);

      elsif (sw(14) = '1') AND (sw(13) = '0') AND (hPos = rightCoordinate) AND (vPos =
bottomCoordinate) then
         addra <= std_logic_vector(TO_UNSIGNED(hImage - 1, addra'length));
      elsif (sw(14) = '1') AND (sw(13) = '0') AND (hPos = rightCoordinate) AND (display = '1')
then
         addra <= std_logic_vector(TO_UNSIGNED(TO_INTEGER(unsigned(addra)) + hImage *
2 - 1, addra'length));
      elsif (sw(14) = '1') AND (sw(13) = '0') AND (display = '1') then
         addra <= std_logic_vector(unsigned(addra) - 1);

      elsif (sw(14) = '0') AND (sw(13) = '1') AND (hPos = rightCoordinate) AND (vPos =
bottomCoordinate) then
         addra <= std_logic_vector(TO_UNSIGNED(hImage * vImage - hImage, addra'length));
      elsif (sw(14) = '0') AND (sw(13) = '1') AND (hPos = rightCoordinate) AND (display = '1')
then
         addra <= std_logic_vector(TO_UNSIGNED(TO_INTEGER(unsigned(addra)) - hImage * 2
+ 1, addra'length));
      elsif (sw(14) = '0') AND (sw(13) = '1') AND (display = '1') then
```

```
            addra <= std_logic_vector(unsigned(addra) + 1);

        elsif (sw(14) = '1') AND (sw(13) = '1') AND (hPos = rightCoordinate) AND (vPos =
bottomCoordinate) then
            addra <= std_logic_vector(TO_UNSIGNED(hImage * vImage - 1, addra'length));
        elsif (sw(14) = '1') AND (sw(13) = '1') AND (display = '1') then
            addra <= std_logic_vector(unsigned(addra) - 1);
        end if;

    end if;
end process;

end VGA_arch;
```

Lisa 4. VHDL koos mälust loetud pildi hägustamiseks

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

-- Kasutatud allikas: https://forum.digilentinc.com/topic/17598-display-image-using-vga-from-block-ram/
-- Kasutatud allikas: https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/

entity VGA is
    Port ( clk : in STD_LOGIC;
           hSync : out STD_LOGIC;
           vSync : out STD_LOGIC;
           sw : in STD_LOGIC_VECTOR (15 downto 0);
           rgb : out STD_LOGIC_VECTOR (11 downto 0));
end VGA;

architecture VGA_arch of VGA is

-- VGA
signal clk_div : STD_LOGIC := '0'; -- 25MHz kell
signal hPos : integer := 1; -- muudetava piksli koordinaadid
signal vPos : integer := 1;
signal display : boolean := false; -- kas pilti edastada

constant hDisplay : integer := 640; -- ekraani laius
constant hLeftBorder : integer := 48; -- ekraani back porch
constant hRightBorder : integer := 16; -- ekraani front porch
constant hRetrace : integer := 96; -- mingi nihe

constant vDisplay : integer := 480; -- ekraani kõrgus
constant vTopBorder : integer := 10; -- ekraani front porch
constant vBottomBorder : integer := 33; -- ekraani back porch
constant vRetrace : integer := 2; -- mingi nihe

-- Pilt
constant hImage : integer := 32; -- pildi laius
constant vImage : integer := 32; -- pildi kõrgus
constant topCoordinate : integer := vDisplay / 2 - vImage / 2; -- pildi ülemise serva koordinaat
constant bottomCoordinate : integer := vDisplay / 2 + vImage / 2; -- pildi alumise serva koordinaat
```

```
constant leftCoordinate : integer := hDisplay / 2 - hImage / 2; -- pildi vasaku serva koordinaat
constant rightCoordinate : integer := hDisplay / 2 + hImage / 2; -- pildi parema serva koordinaat

-- Maatriks
type matrix is array (1 to vImage, 1 to hImage) of std_logic_vector(11 downto 0);
signal image : matrix;
signal firstTime : boolean := true;
signal i : integer := 1;
signal j : integer := 1;

-- Keskmistamine
constant kernel : integer := 3;
constant halfKernel : integer := kernel / 2;

-- BRAM
signal wea : STD_LOGIC_VECTOR(0 downto 0) := "0";
signal addra : STD_LOGIC_VECTOR(9 downto 0) := (others => '0');
signal dina: STD_LOGIC_VECTOR(11 downto 0) := (others => '0');
signal douta : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');

component image_ram is
   PORT (
        clka : IN STD_LOGIC;
        wea : IN STD_LOGIC_VECTOR(0 downto 0);
        addra : IN STD_LOGIC_VECTOR(9 downto 0);
        dina : IN STD_LOGIC_VECTOR(11 downto 0);
        douta : OUT STD_LOGIC_VECTOR(11 downto 0));
end component;

begin

-- BRAM
U1: image_ram Port Map (clka=>clk, wea=>wea, addra=>addra, dina=>dina, douta=>douta);

process(clk_div)
begin
if rising_edge(clk_div) AND firstTime then
   image(i, j) <= douta;
   addra <= std_logic_vector(unsigned(addra) + 1);
   if (i = vImage) AND (j = hImage) then
      firstTime <= false;
   elsif (j = hImage) then
      i <= i + 1;
      j <= 1;
   else
      j <= j + 1;
   end if;
end if;
end process;
```

```
-- 25 MHz kell
process(clk)
variable clk_counter : STD_LOGIC := '0';
begin
   if rising_edge(clk) then
      clk_counter := not clk_counter;
      if clk_counter = '0' then
         clk_div <= not clk_div;
      end if;
   end if;
end process;

-- Piksli asukoha horisontaalne muutmine
process(clk_div)
begin
   if rising_edge(clk_div) then
      if hPos = hDisplay + hLeftBorder + hRightBorder + hRetrace then
         hPos <= 0;
      else
         hPos <= hPos + 1;
      end if;
   end if;
end process;

-- Piksli asukoha vertikaalne muutmine
process(clk_div)
begin
   if rising_edge(clk_div) then
      if hPos = hDisplay + hLeftBorder + hRightBorder + hRetrace then
         if vPos = vDisplay + vTopBorder + vBottomBorder + vRetrace then
            vPos <= 0;
         else
            vPos <= vPos + 1;
         end if;
      end if;
   end if;
end process;

-- Horisontaalse sünkronisatsiooni signaal
process(clk_div)
begin
   if rising_edge(clk_div) then
      if (hPos <= hDisplay + hRightBorder) OR (hPos > hDisplay + hRightBorder + hRetrace) then
         hSync <= '1';
      else
         hSync <= '0';
      end if;
   end if;
end process;
```

```
-- Vertikaalse sünkronisatsiooni signaal
process(clk_div)
begin
   if rising_edge(clk_div) then
      if (vPos <= vDisplay + vTopBorder) OR (vPos > vDisplay + vTopBorder + vRetrace) then
         vSync <= '1';
      else
         vSync <= '0';
      end if;
   end if;
end process;

-- Kunas pilti edastada
process(hPos, vPos)
begin
   if (hPos <= rightCoordinate) AND (hPos > leftCoordinate) AND (vPos <= bottomCoordinate)
AND (vPos > topCoordinate) then
      display <= true;
   else
      display <= false;
   end if;
end process;

-- Edastatav pilt
process(clk_div)
variable r : integer := 0;
variable g : integer := 0;
variable b : integer := 0;
variable kernelSum : integer := 0;
variable vStart : integer := 0;
variable hStart : integer := 0;

begin
   if rising_edge(clk_div) then
      if display then
         if sw(15) = '0' then
            rgb <= image(vPos - topCoordinate, hPos - leftCoordinate);
         elsif sw(15) = '1' then
            r := 0;
            g := 0;
            b := 0;
            kernelSum := 0;
            vStart := vPos - topCoordinate - halfKernel;
            hStart := hPos - leftCoordinate - halfKernel;
            for c in 0 to kernel - 1 loop
               for d in 0 to kernel - 1 loop
                  if (vStart + c > 0) AND (vStart + c <= vImage) AND (hStart + d > 0) AND (hStart +
d <= hImage) then
                     r := r + to_integer(unsigned(image(vStart + c, hStart + d)(3 downto 0)));
                     g := g + to_integer(unsigned(image(vStart + c, hStart + d)(7 downto 4)));
```

```
                    b := b + to_integer(unsigned(image(vStart + c, hStart + d)(11 downto 8)));
                    kernelSum := kernelSum + 1;
                end if;
            end loop;
        end loop;
        rgb <= std_logic_vector(to_unsigned(b / kernelSum, 4)) &
std_logic_vector(to_unsigned(g / kernelSum, 4)) & std_logic_vector(to_unsigned(r / kernelSum,
4));
      end if;
    else
      rgb <= (others => '0');
    end if;
  end if;
end process;

end VGA_arch;
```

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Oliver Oll

1. Annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

“Õppeaine „Digitaalne loogika“ praktikumide täiendamine riistvaraliste paralleelülesannetega”

mille juhendaja on Margus Rosin

reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace’i kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuand- mete kaitse õigusaktidest tulenevaid õigusi.

Oliver Oll

20.05.2022