TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAY40LT

Romet Kõiv 120437

DIGILENT NEXYS 4 PLATVORMI RAKENDAMINE UJUPUNKTARVUDE TEHETE REALISEERIMISEKS XILINX VIVADO KESKKONNAS

bakalaureusetöö

Juhendaja: Priit Ruberg
Magister
Nooremteadur

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Romet Kõiv

24.05.2016

ARVUTITEHNIKA INSTITUUT
TALLINNA TEHNIKAÜLIKOOL

BAKALAUREUSETÖÖ ÜLESANNE

Üliõpilane: **ROMET KÕIV** Matrikkel: **120437**

Lõputöö teema eesti keeles:
Digilent Nexys 4 platvormi rakendamine ujupunktarvude tehete realiseerimiseks Xilinx Vivado keskkonnas

Lõputöö teema inglise keeles:
Implementing Floating-Point Operation Algorithms On Digilent Nexys 4 Development Board Using Xilinx Vivado

Juhendaja (nimi, töökoht, teaduslik kraad, allkiri):
Priit Ruberg, Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutitehnika instituut, Arvutisüsteemide projekteerimise õppetool , Nooremteadur

Konsultandid:

Lahendatavad küsimused ning lähtetingimused:
1. Mõista ujupunktkoma arvude kasutamise vajalikkusest, võimalustest ning piirangutest
2. Õppida tundma Nexys 4 platvormi võimalusi
3. Uurida Xilinx Vivado programmeerimiskeskkonda ning koostada üksikasjalik juhend keskkonnas orienteerumiseks (üks võimalik juhendi näide)
4. Koostada ujupunktkoma arvude liitmise, lahutamise ja korrutamise realiseerimise algoritmid
5. Realiseerida tehted VHDLis - simulatsioon kui ka süntees FPGAle
    1. Sisendite normaliseerimine
    2. Tehte teostamine
    3. Väljundi normaliseerimine (kui vajalik)
6. Verifitseerida tulemused

Eritingimused: Negatiivsed arvud on esitatud täiendkoodina. Formaat üks bitt märk, 12 bitti mantiss ja seitse bitti astendaja. Tehete teostamisel ei kasutata DSP moodulit.

Nõuded vormistamisele: Vastavalt Arvutitehnika instituudis kehtivatele nõuetele

Lõputöö estamise tähtaeg: 03.06.2016

Ülesande vastu võtnud: _____kuupäev: 11.03.2016
(lõpetaja allkiri)

Annotatsioon

Lõputöö on kirjutatud viies osas. Algab ujupunktarvude tutvustamisega ning seletab lahti töös kasutatavate ujukomatehete valemid. Järgneb seadme tutvustus, jälgides vaid töös vajaminevaid sisendeid ja väljundeid. Autor kirjutab lahti formaadi, mis on töö aluseks ja kuidas teha andmete kuvamine ning sisestamine. Pikemalt on juttu ka programmi sisendite ja väljundite sidumisest konkreetse seadmega.

Ujukomatehetest on käsitletud liitmist, lahutamist ja korrutamist, mis on realiseeritud eraldi programmipaketiga. Autor rõhutab ujupunktarvude normaliseerimise vajalikkust, viies läbi formaadi kontrolli enne ja pärast tehete realiseerimist. Neljandas osas võtab autor ette valminud programmi testimise, milles käsitleb komponentide individuaalset kontrolli. Samuti on välja toodud võtted terviku testimiseks.

Viiendas Peatükis kirjutab autor versioonihalduse võimalustest. Tööle on lisatud juhendid kuidas alustada tööd Xilinx Vivado programmiga kasutades graafilist keskkonda ning teostada samad toimingud käsurealt. Käsurea juhendi juurde kuulub teadmusbaas versioonihalduse integreerimisest Xilinx Vivado projekti.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 34 leheküljel, 5 peatükki, 24 joonist, 3 tabelit.

Abstract

Implementing floating-point operation algorithms on Digilent Nexys 4 development board using Xilinx Vivado.

Thesis starts with theory about floating point numbers – what they are and possible benefits of using them. Theory compares different formats representing binary vectors and gets to conclusion that floating point numbers use economical method to handle decimal numbers with wide range. One goal of this thesis is to program working sample of VHDL code, what can do addition, subtract and multiply of two floating point number. Second goal is to provide Xilinx Vivado guide for other students as most of the newer examples based on Verilog language.

Chapter 3 contains detailed description about switches and led selections on the board and how to link them using constraint file. Author decided to put output to eight digit seven segment display and explained all steps what are needed to get display properly working. In addition all handled input combination are described.

Chapter 4 puts focus on floating point calculation logic. Due thesis restrictions, author was not able to use DSP module for calculations. Before calculations, normalizing mantissa VHDL code explained in details. Subtract solved by adding complement binary number instead of positive. Addition itself uses special formula from Chapter 2. Multiplication is done using signed variables converted to vectors. As multiplication result format is two times longer than allowed, normalizing output of mantissa applied.

Author explains testing methods used in this thesis to prove that programmed solution meets expectations. In addition version control also explained and some hints given to make VHDL programming modern. Inside extras guide to create project with Vivado using script is explained.

The thesis is written in Estonian and contains 34 pages of text, 5 chapters, 24 figures, 3 tables.

Lühendite ja mõistete sõnastik

ALU	*Arithmetic logic unit*
ATI	TTÜ Arvutitehnika instituut
Downto	*Loendab ülevalt alla, loogilise vektori esitus*
DSP	*Digital signal processor*
FPGA	*Field-programmable gate array*
GIT	*Global information tracker*
HDL	*Hardware description language*
LED	*Light-emitting diod*e
LUT	*LookUp table*
RTL	*Register-transfer level*
Slice	*Logic slice*
VHDL	*VHSIC Hardware description language*
TCL	*Tool command language*

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Arvutitehnika instituudis loetavates ainetes Digitaalsüsteemid (IAY0150) ja Programmeeritavad loogikaskeemid (IAF0520) praktilise osa realiseerimisel on kasutusel riistvarakirjelduse (HDL) realisatsiooni keelena VHDL. Arendustarkvara, mille abil luua, simuleerida ning rakendada väliprogrammeeritaval maatriksil (FPGA) toimivat koodi pakub mitu tootjat. Praktikumides on tudengil kokkupuude Xilinx ISE keskonna ja Modelism tarkvaraga.

Bakalaureuseõppe tasemel, kasutatakse aines Digitaalsüsteemid näidetena Xilinx ISE programmi, mille abil disainida ja sünteesida FPGA tarkvara. Uuemate seadmete nagu Nexys 4 koodinäited on pakutud enamuses Xilinx Vivado programmi ja Verilog keele baasil. Oma töös uurib autor, kuidas õnnestub varasemat praktikat VHDL keeles, rakendada Nexys 4 platvormil kasutades Xilinx Vivado arendustarkvara. Register-siirete tasemel (RTL), VHDL keeles realiseerib autor ujupunktkarvude liitmise, lahutamise ja korrutamise aine Arvutite aritmeetika ja loogika (IAY0140) teoreetilise osa baasil seadmel Digilent Nexys 4. RTL abstraktsiooni tase ei eelda loogikakomponentide füüsilist kirjeldusi ja piisab vaid algebralisest lähenemisest.

Töö üheks eesmärgiks oleva ujukomatehete programmipaketile kirjutab autor juurde seadme indikaatorile tulemuste kuvamise ja operandide lugemise alamprogrammid. Samuti kasutab autor oma töös versioonihaldustarkvara võimalusi, mille omadusi ja võimalusi tutvustab eraldi peatükis.

Autor lisas tööle Xilinx Vivado kasutusjuhendi, tuues välja punktid, mis võivad uue projektiga töötamisel kitsaskohaks osutuda. Antud tööle on lisatud ka juhend, kuidas alustada tööd Xilinx Vivado programmiga kasutades Tcl skripti keelt, mis on toetatud ka IAY0150 ja IAF0520 praktikumides kasutusel olevas ISE versioonis 14.7.

2 Töö teoreetilised alused

Arvude esitamine kahendkujul seab piirangud arvu väärtuse suurusele. Kui valida näiteks 8-bitine register, siis positiivseid täisarve saab sellega esitada väärtusega $0_{10...}256_{10}$ [4] . Reaalarvude esitamisel tekib märgi ja koma küsimus. Märgiga täisarve saab 8-bitises registris esitada vahemikus $-65_{10}...127_{10}$. Kinnispunktarve saab esitada vaid siis kui on kokku lepitud, mitu kohta on ette nähtud enne ja pärast koma. Näiteks võtame 8-bitisel skaalal märgiks 1-bitt enne koma 4-bitti ja pärast koma 3-bitti. Nii suudame esitada arve vahemikus -9.875_{10} ... 15.875_{10}. Antud näide illustreerib kui palju mäluruumi kulub formaadi säilitamisele.

Lahenduseks on ujupunktarvud. Ujupunktarv on kahendsüsteemi arvu kaheosaline esitusviis, mis koosneb mantissist ja astendajast. Ujupunktarvu väärtus avaldub valemiga (1).

$$mantissa * 2^{astendaja} \quad (1)$$

Mantiss on puhtmurdarv ja astendaja on täisarv. Mantissi märk on kogu ujupunktarvu märk. Arvutamise teel saadud arvu väärtus võimaldab esitada arve suure diapasooniga skaalal. Ujupunktarvu mantiss avaldatakse alati normaliseeritud kujul, ainult nii saab arvutustes kindel olla, kus on koma. Positiivse normaliseeritud mantiss esitatakse kujul, et väärtus jääb vahemikku $0.1000..._2 <= m <= 0.11111..._2$ ehk $0.5_{10} <= m < 1_{10}$. Negatiivne normaliseeritud mantiss esitatakse kujul $1.00000..._2 <= m <= 11.01111..._2$ ehk $-1_{10} <= m < -0.5_{10}$ [1] .

Mantissi korrutamine arvu kaks astmega väljendub loogikaskeemides nihutamisega. Negatiivne astendaja viib koma vasakule ja vähendab mantissi väärtust. Positiivne astendaja liigutab koma paremale ja muudab mantissi väärtuse suuremaks.

Algselt kasutatud näites toodud kaheksa bitisel registril valides ühe biti märgiks, neli mantissiks ja kolm astendajaks (astendaja on alati positiivne), saab esitada arve vahemikus -72_{10} ... 120_{10} , mis on tunduvalt suurem kui kinnispunktarvu puhul.

Oma töös kasutan järgnevat 20-bitist formaati, millest vasakult paremale üks bitt on märk, seitse bitt astendaja (millest üks bitt on märk) ja 12-bitti mantiss. See võimaldab tehteid arvudega, mille väärtus jääb vahemikku -131136 ...262080.

2.1 Ujupunktarvude liitmine ja lahutamine

Normaliseeritud ujupunktarvude A ja B liitmine toimub valemiga (2), kui A astendaja on suurem kui B astendaja. Ujupunktarvud A, B ja C on esitatud valemis (1) esitatud kujul, mis liitmise valemis tähendab, et peame jagame mõlemad liidetavad läbi, kaks astmes P_A-ga. Taolisel viisil oleme saanud ujupunktarvuna esitatavale summale astendaja. Uue mantissi arvutame välja allesjäänud avaldisest.

$$C = A + B = m_A \times 2^{p_A} + m_B \times 2^{p_B} = 2^{p_A}\left(m_A + m_B \times 2^{p_B - p_A}\right) \tag{2}$$

Kui B astendaja on suurem kui A astendaja, siis tulevad liidetavad jagada läbi kahe astendaja P_B-ga. Tulemuseks saame valemi (3). Taoline lähenemine on vajalik, et sulgudes oleva arvutuse tulemusena saadud mantiss ei läheks formaadist välja. Otsustades enne liitmist, kumb astendaja on suurem, tagame, et arvutatav mantiss saab olema formaadis. Võrdsete astendajate puhul pole valemi valik oluline.

$$C = A + B = m_A \times 2^{p_A} + m_B \times 2^{p_B} = 2^{p_B}\left(m_B + m_A \times 2^{p_A - p_B}\right) \tag{3}$$

Lahutamisel kasutame täiendkoodi liitmist, valem (4).

$$C = A - B = B + (-B) \tag{4}$$

2.2 Ujupunktarvude korrutamine

Kuna astendaja alus on kaks, siis kasutame teadmist, et mantissid korrutame ja astendajad liidame. Korrutis ja korrutatav on kujul valem (1). Korrutamise valem on järgnev

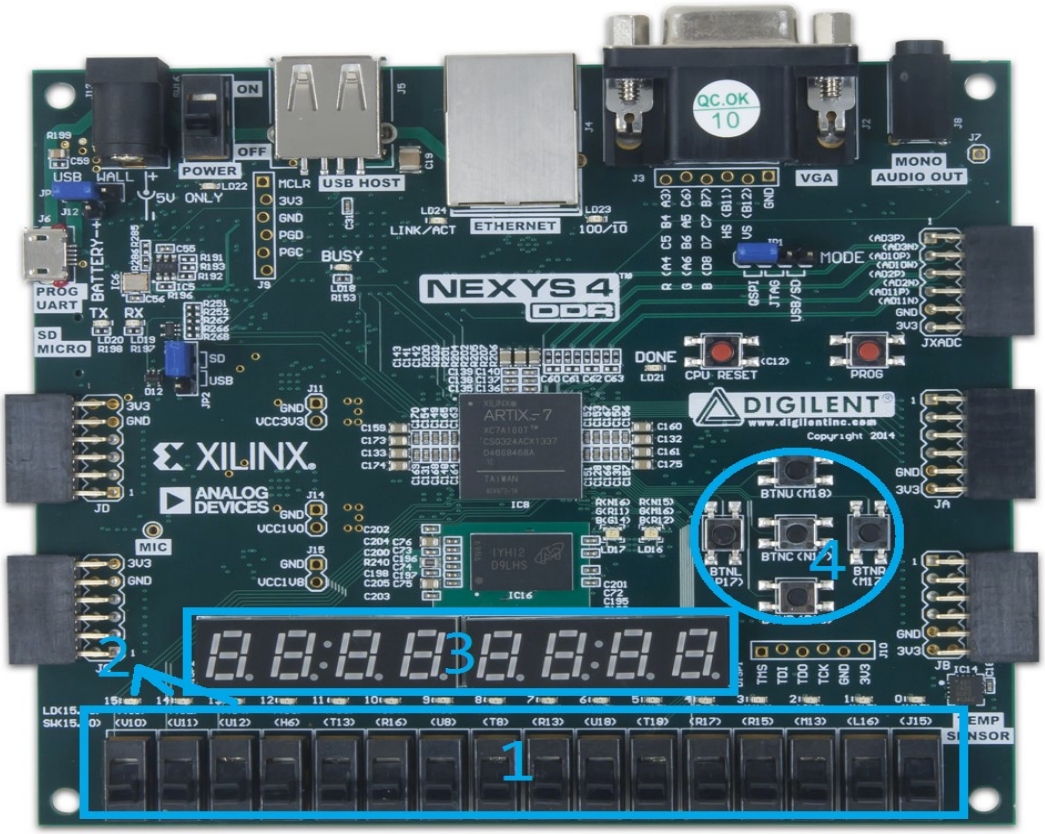
$$C = A \times B = m_A \times m_B \times 2^{p_A + p_B} \tag{5}$$

Korrutamisel on tarvilik arvestada, et korrutise mantiss võib olla enne tulemuse normaliseerimist kaks korda sama pikk kui korrutatavad.

3 FPGA Digilent Nexys 4 sidumine programmiga

Digilent Nexys 4 arendusplatform baseerub firma Xilinx Artix-7™ [7] perekonna digitaalsel programmeritaval loogikal. Omadustes on kirjas, et põhikiip sisaldab 15850 loogilist viilu (*slice*), millel igaühel on neli 6 sisendiga LUT ja 8 triggerit. Lisaks saab kasutada mälu 4860 Kbit RAM. Seadmel on ka 240 DSP viilu. Taktigeneraator töötab sagedusel 450 MHz [2] . Arvestades töö mahulise piiranguga ja autori omanduses olevate tarkvarakomponentide hulgaga, otsustas autor loobuda seadme poolt pakutava 240 DSP48E1 võimalustest. Mooduli DSP48E1 dokumentatsioon lubab efektiivselt sooritada liitmist ja korrutamist, sest omab vastavat ALU komponenti, kuid eeldab sisendite, väljundite ja käitumise põhjalikku tundmist.

Autor toob välja töös kasutatavad sisendid ja väljundid plaadilt (Joonis 1 ja Tabel 1).



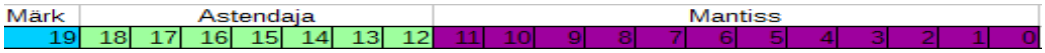
Joonis 1. Digilent Nexys 4

Tabel 1. Joonis 1 numbrite tähendused ja seotud muutujad.

#	**Selgitus**	#	**Selgitus**
1	16 Lülitit (SW)	3	7-segmendi indikaator (SSEG_CA, SSEG_AN)
2	16 LED'i, tavalised tuled (LED)	4	Viis surunuppu (BTN)

3.1 Programmis kasutatav andmete formaat

Oma töös kasutan 20-bitist formaati. Vasakult paremale lugedes 1-bitt on märk, 7-bitt astendaja ja 12-bitt mantiss (Joonis 2). Astendaja esimene bitt kuulub märgile. Digilent Nexys 4 7-segmendi indikaatoril kuvan astendaja ja mantissi kümnendesituse.



Joonis 2. Ujupunktarvu formaat

Mantissi sisestamine eeldab, et koma on paigas ja sisestatakse vaid number pärast koma. See toob kaasa olukorra, kus sisestatud vektor pole normaliseeritud. Esialgsel kasutusel tekib küsimus, miks kuvatakse ekraanil arvu väärtus, mitte murdarvu väärtus. Murdarvu väärtuse määrab arvu kaks astendaja ja pärast koma on need negatiivsed. Autor seadis väljundi programmeerimisel, ekraanile kuvatava arvu pikkuse piirangu, milleks on neli kohta. Töö formaadi tõttu on kõige suurem arv, mida saab kuvada ekraanil 4095. Näiteks positiivse $0{,}111_2$ esitus ekraanil peaks olema $0{,}875_{10}$, mis teeb kinnispunktarvu esitamise maksimaalseks pikkuseks kolm kohta ja muudab pikema mantissi esituse ebatäpseks.

Astendaja sisestamisel ja kuvamisel on ekraani väärtus võrdne kahendvektoriga. Programm väljastab tulemused vaid normaliseeritud kujul. Näiteks vektori $0{,}1000000000_2$ väärtus on 2048_{10} (Joonis 22).

3.2 Sidumine füüsilise plaadiga

Tarkvaraarendus keskkonnas Xilinx Vivado tuleb VHDL projektiga ühendada piirangute fail (Nexys4_VHDL.xdc). Piirangute faili eesmärk on siduda VHDL muutujad sisendite ja väljunditega ning kirjeldada sünteesile konkreetse seadme võimalusi. Joonis 3 selgitab, kuidas siduda loogilise vektori SW elemendid lülititega. Segmendi sisu on

määratud muutujaga *SSEG_CA*, mille andmetüüp on *std_logic_vector* (edaspidi vektor). Segmenti, millele sisu saadetakse valitakse anoodi vektoriga *SSEG_AN*.

```
#Bank = 34, Pin name = IO_L21P_T3_DQS_34,              Sch name = SW0
set_property PACKAGE_PIN U9 [get_ports {SW[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]
#Bank = 34, Pin name = IO_L2N_T0_34,                   Sch name = CA
set_property PACKAGE_PIN L3 [get_ports {SSEG_CA[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {SSEG_CA[0]}]
#Bank = 34, Pin name = IO_L18N_T2_34,                  Sch name = AN0
set_property PACKAGE_PIN N6 [get_ports {SSEG_AN[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {SSEG_AN[0]}]
```

Joonis 3. Lülitite sidumine muutujaga SW.

Sama loogikaga on seotud Peatükis 3 loetletud töös kasutatavad sisendid ja väljundid (Tabel 1). Kasutusel on vaid kommenteerimata read.

3.3 VHDL komponente ühendav pealmine disain

Autor uurib peamist komponenti, mis seob kokku sisendi, väljundi ja ujukomatehted. Programmis on kaks protsessi: nupuvajutuse registreerimine ja nupuvajutuse töötlemine. Selleks, et seadmega oleks võimalik andmeid kuvada ja sisestada, tuleb kasutada tundlikkuse nimistut [6] (lk 38-40). Tundlikkuse nimistu on jälgitavate signaalide loend protsessil, mille väärtuse muutumine käivitab protsessi.

Nupuvajutuse registreerimise protsess ei täida töö eesmärki kui tundlikkuse nimistus on vaid *clk* signaal. Sooritatava tehte seadmiseks on tarvis registreerida eelnev ja praegune nuppude kombinatsioon. Tehe registreeritakse vaid siis, kui on olemas signaalid: registreeritud nupuvajutus, loetud nupuvajutus ja protsessori takti tõus (Joonis 4). Antud kombinatsioon registreerib nupuvajutuse ja režiimi vektori.

```
btn_reg_process : process (clk,btnDeBnc,btnReg)--CLK
   begin
     --Kui on tõusev front ja nupp btnDeBnc pole null
     if  rising_edge(CLK) and btnDeBnc /="00000" then
         btnReg <= btnDeBnc;
         case btnReg is
         when "10000" => mode <="00"; --liitmine
         when "01000" => mode <="01"; --lahutamine
         when "00100" => mode <="10"; --korrutamine
         when others => mode <="11"; -- ei toimu midagi
         end case;
     end if;
end process;
```

Joonis 4. Nupuvajutuse registreerimise protsess.

Tabel 2. Kasutuses olevate nuppude funktsioonid

Nupp	**Funktsiooni nimi**
BTNR	Lülititel oleva väärtuse sisselugemine ja muutujateks salvestamine
BTNC	Liitmine ja tulemuse ekraanil kuvamine
BTNL	Lahutamine ja tulemuse ekraanil kuvamine
BTNU	Korrutamine ja tulemuse ekraanil kuvamine
BTND	Nullib muutujad ekraanid

Nupuvajutuse registreerimisel on kasutatud komponenti *debouncer,* mis on võetud Nexys 4 näidiseprojektist GPIO [5]. Taktigeneraator töötab miljonites kordades kiiremini kui tegutseb inimene, mille tõttu tuleb tagada, et registreeritakse vaid ühekordne signaali muutus. *Debouncer* on seatud kontrollima signaali 65536 takti järel. Võimaldab korraga jälgida mitut signaali, mis antud töös võimaldab kasutada muutujana kombineeritud nuppude vektorit.

3.3.1 Muutjate sisestamine ja kuvamine

Vastavalt lülitite 14 ja 13 asendile (Tabel 3), loetakse muutujad vastavasse registrisse. Ekraanil kuvatakse sisestuse ajal kahendnumber kümnendkujul. Sisestuse registreerib BTNR vajutus. Kui registris on varem registreeritud väärtus, siis kirjutatakse see üle. Kasutajat ei teavitata toimingu õnnestumisest. Programmiga seatakse kõigile muutujatele vaikimisi väärtus null.

Kaheksast indikaatorist esimesel kahel näitame koode „M1”, „P1”,”M2”„P2” vastavalt SW (14 *downto* 13) koodile. Järgneb tühi indikaator ja järgmised neli indikaatorit kuvavad kahendkoodi kümnend kujul. Samuti registreeritakse muutujad (Joonis 5). Jooniselt võib näha, et on kasutatud kahte ekraanile kuvamise VHDL muutujat *numbers* ja *onscreen*. Tegelikult kirjutatakse *onscreen* muundatud väärtus hiljem *numbers* vektori lõppu, millest tuleb juttu Peatükis 3.5. Muunduri sisendvektori pikkus on 12 ühikut, seega kirjutatakse astendajale vastavalt märgile ette kas ühed või nullid.

```
if btnReg = "00001" and btnDeBnc ="00000" then
   if SW(14 downto 13)="00" then
      numbers(31 downto 28):="1010"; --M
      numbers(27 downto 24):="0001"; --1
      mant_a<=SW(15)&SW(11 downto 0); --mantiss A
      onscreen<=mant_a;    --mantiss A ekraanile
   elsif SW(14 downto 13)="01" then
      numbers(31 downto 28):="1011"; --P
      numbers(27 downto 24):="0001"; --1
      pow_a<=SW(15)&SW(6 downto 0); -- A astendaja märgiga
      onscreen(12)<=pow_a(7); --Astendaja märk ekraanile
      onscreen(6 downto 0)<=pow_a(6 downto 0); --Astendaja
      if pow_a(7)='1' then --Negatiivne
         onscreen(11 downto 7)<="11111"; --Märk täidab puuduva
      else --Positiivne
         onscreen(11 downto 7)<="00000"; --Märk täidab puuduva
      end if;
  end if;
--Peamine If jätkub järgmise kombinatsiooniga
```

Joonis 5. Operandi A sisestamine.

Erinevalt surunuppudest võivad samadel lülititel olla erinevad funktsioonid funktsioonid (Tabel 3).

Tabel 3. Lülitite tähendused

SW	**Funktsioon**
15	Märk. 0 siis „+” ja 1 „-”
14-13	Muutuja tüüp sisendis „00” - mantiss A „01” - astendaja A „10” - mantiss B „11” - astendaja B Muutuja tüüp väljundis „00” - mantiss C „01” - astendaja C
11-0	Muutuja väärtuse lugemine. Astendaja vaid 6-0

3.4 Väljund 7-segmendilisel indikaatoril

Võimalik on kasutada kaheksakohalist 7-segmendilist indikaatorit. Disainis pole segmentidel mälu, mis hoiaksid neile pandud väärtusi. Muutujaid jälgides on näha, et

korraga saab kasutada vaid ühe segmendi sisu ja selle valimise vektorit. Antud loogika lubab näidata sama väärtust valitud ekraanil. Inimese silm pole võimeline eristama piisavalt kiiret vilkumist, mille tõttu võib kogu väärtust kanda ekraanile tsükli kaupa. Kasutan selleks komponenti „*segmentdriver*".

```
component segmentdriver is
    Port ( display : in STD_LOGIC_VECTOR (31 downto 0);
           seg : out STD_LOGIC_VECTOR(7 downto 0);
           select_Display : out STD_LOGIC_VECTOR(7 downto 0);
           clk : in STD_LOGIC);
end component;
```

Joonis 6. Komponent segment draiver muutujad.

Segmentdriver koosneb veel omakorda komponentidest „clock_divider" ja „*segmentdecoder*". *Clock_divideri* ülesanne on „aeglustada" tavalist taktigeneraatorit, selleks loendab ta takti ja liidab tulemuse vektorisse pikkusega 16-bitti. Antud võte vahetab anoodi järgmisele indikaatorile nii, et näitab ühele sama väärtust 65535 tsüklit, mis on ikkagi vaid 1/6866 sekundit 450MHz juures.

```
process(slow_clock) -- Aeglasema taktiga näitame valime indikaatori
    variable display_selection : STD_LOGIC_VECTOR(2 DOWNTO 0);
    begin           -- kui aeglane takt muutub.
    if slow_clock'event and slow_clock = '1' then
    case display_selection is    --  milline indikaator
    when "000" => temporary_data <=display(3 downto 0);
        select_display <= "11111110"; -- viimane indikaator
        display_selection := display_selection + '1';
```

Joonis 7. Segment draiveri tsükkel.

Segmentdecoderi ülesanne on 4-bitt sisendvektor tõlkida visuaalseks kujutiseks. Segmendi ja indikaatori valimine toimub vastupidise bitiga – väärtus „0" on aktiveerija. Kasutusel on numbrid 0-9 ja sümbolid. Joonis 8 tõlgendab sümbolite kombinatsioonid ja neile vastava kujutise indikaatoril, numbrid kuvatakse vastavalt kahendkoodi väärtusele.

```
when "1010" => Decode_Data := "00010101"; --M
when "1011" => Decode_Data := "01100111"; --P
when "1100" => Decode_Data := "11111110"; --null ja punkt
when "1101" => Decode_Data := "00000001"; --miinus
when "1110" => Decode_Data := "00110001"; --miinus 1
when others => Decode_Data := "00000000"; --Tühi
```

Joonis 8. Indikaatorile kuvatavad sümbolid.

3.5 Kahend-kümnend muundur

Muunduri eesmärk on lõhkuda kindla väärtusega sisendvektor, nii et moodustub väljund, mida tohib tõlgendada 4-bitiste alamvektorite kattumatu hulgana. Ühe 4-bitise vektori väärtus ei tohi ületada numbrit üheksa. Kasutatud nihe vasakule, toob tühjale kohale väärtuse „0”.

Võtame näiteks nullidega töidetud 12 kohalise vektori, mille lõppu lisame koodi 1111_2 ning hakkame seda vasakule nihutama, nii mitu korda kui on lisatud kahendkoodi formaadi pikkus. Tulemuseks loeme kaks 4-bitt vektorit vasakult ning algse sisendi jätame lugemata. Tagastavate vektorite väärtusi kontrollime tsükli jooksul madalamast järgust kõrgemale (paremalt vasakule). Kui tsükli alguses 7 *downto* 4 vahemikus olev kahendarv on saanud väärtuse viis või enam, siis liidame talle kolm, mis lisab ületäituvuse kõrgemasse järku.

Selgub, et see juhtub kui 0111_2 on juba vektoris ja 1000 on veel ootel. Pärast kolme lisamist saab 7 *downto* 4 koodiks 1010_2. Tsükli lõpus toimub nihe vasakule. Nihke järel on väljundvektoritel uued väärtused: 11 *downto* 8 saab koodiks 0001_2 (1_{10}) ja 7 downto 4 0101_2, mis on väärtusega viis. Kuna tsükkel saab läbi, siis vektorid säilitavad oma uued väärtused, vastavalt üks ja viis ehk ekraanil „15”.

Joonis 9 kujutab eelnevas kahes lõigus kirjeldatud näite realisatsiooni. Programmist on välja jäetud järgud kümme, sada ja tuhat, kuna kood hakkab korduma.

```
bin_to_bcd : process (number,sign,clk)
  variable shift : unsigned(27 downto 0); -- nihkevektor
  variable negative : unsigned(11 downto 0);  -- negatiivne arv
  -- järgud
  alias num is shift(11 downto 0);  --ühesed
  alias one is shift(15 downto 12);  --kümnesed
  begin
  num := unsigned(number);
  --Kuni numbri pikkuseni
  for i in 1 to num'Length loop
  if one >= 5 then --  Ühesed suuremad kui 5
    one := one + 3; -- lisame kolm
  end if;
  -- Tsükli lõpus nihe vasakule
   shift := shift_left(shift, 1);
 end loop;  -- viimasena loeme väärtuse
 result(3 downto 0)     <= std_logic_vector(one);
```

Joonis 9. Kahend-kümnend muundur.

4 Ujupunktarvude tehete realiseerimine

Kõik ujukomatehted on programmeeritud ühte moodulisse nimega *float_calc,* mille struktuur on kujutatud Joonisel 10. Sisenditena on arvu A mantiss (*mantA_in*) ja astendaja (*powA_in*) ning arvu B mantiss (*mantB_in*) ja astendaja (*powB_in*). Algses koodis sai kasutatud muutujaid ilma „_in" lisandita. Töö käigus selgus, et programmi loogikat on otstarbekam programmeerida laiendades mantissi vektorit vasakult ühe biti võrra. Edasistes funktsioonides tähistavad muutujad algusega *mant* mantissi ja *pow* – astendajat. Teatud operandide kombinatsiooni korral tagastab ujukomaarvutus veakoodi, mille väärtused on kommentaarina välja toodud programmi päises (Joonis 10). Oluline sisend on teostatav tehe (*mode),* mis vastab Peatükis 3.3 sätestatud tingimustele. Kombinatsioon „11" on jäetud vabaks. Joonise lõpus on deklareeritud vektorite maksimaalsete pikkuste konstandid, mida kasutan kogu mooduli ulatuses.

```
--Veakoodid
--1001 Positiivset tulemust ei saa vähendada
--1010 Negatiivsed tulemust ei saa vähendada
--1111 mantissi ei saa vähendada

entity float_calc is
    Port ( clk : in STD_LOGIC;
           errorCode : out STD_LOGIC_VECTOR (3 downto 0):=(others => '0');
           mantA_in : in STD_LOGIC_VECTOR (12 downto 0):=(others => '0');
           mantB_in : in STD_LOGIC_VECTOR (12 downto 0):=(others => '0');
           powA_in : in STD_LOGIC_VECTOR (7 downto 0):=(others => '0');
           powB_in : in STD_LOGIC_VECTOR (7 downto 0):=(others => '0');
           mode : in STD_LOGIC_VECTOR (1 downto 0);
           mant : out STD_LOGIC_VECTOR (12 downto 0):=(others => '0');
           pow : out STD_LOGIC_VECTOR (7 downto 0):=(others => '0'));

end float_calc;

--Konstandid, mida töös kasutan
variable mlen : INTEGER range 1 to (INTEGER'high) := 13; -- Mantissi pikkus
variable plen : INTEGER range 1 to (INTEGER'high) := 7; -- Astendaja pikkus
```

Joonis 10. Mooduli *float_calc* sisendid ja väljundid koos veakoodidega.

4.1 Operandide mantissi normaliseerimine

Operandide sisestamisel on lubatud kasutada mugavuse mõttes normaliseerimata mantissi. Mantissi väärtused loetakse alati tingimusena null koma ja vastavalt märgile ühed või nullid ning arv ekraanil. Normaliseerimise alguses teeb programm kindlaks, millise märgiga on mantiss. Positiivse mantissi korral otsib esimest ühte vasakult paremale lugedes, mis tuleb pärast märki. Leides esimese nulli, jääb programm seisma. Järgneb normaliseeritud mantissi eraldamine nii, et ta algab koodiga 00 ja edasi on vektor, mis algab leitud esimesest ühest. Kuna tulemuse salvestame uude muutujasse, mille kõik kohad on null, siis pole vaja viimaseid numbreid nulliks kirjutada. Astendajat aga vähendame, kuna otsitud normaalkujul mantiss saab olema suurem kui tema algne väärtus.

Negatiivse mantissi korral tuleb programmil analüüsida vektorit vasakult paremale. Leides pärast märgi kohta esimese nulli, eraldab programm normaliseeritud kuju.

Edasised liigutused sarnanevad positiivse mantissi normaliseerimisele. Astendaja väheneb ja vektor nihkub koma juurde (Joonis 11).

```
variable mantA : STD_LOGIC_VECTOR (13 downto 0):=(others => '0');
--Normaliseerime mantissi A
--Otsime esimese ühe, see võib olla ka viimane
if mantA_in(mlen-1)='1' then
    --Tsükkel on ülevalt alla stiilis
    for index in (mlen-2) downto 0 loop
    if mantA_in(index)='0' then
       --Pikendame uut mantissi
       mantA(mlen downto (mlen-2)-index):="11"&mantA_in(index downto 0);
       powA:= powA-((mlen-2)-index);
       exit;
       end if;
    end loop;
else
    for index in (mlen-2) downto 0 loop
      if mantA_in(index)='1' then
         --Pikendame uut mantissi
         mantA(mlen downto (mlen-2)-index):="00"&mantA_in(index downto 0);
         powA:=powA-((mlen-2)-index);
         exit;
         end if;
      end loop;
end if;
```

Joonis 11. Mantissi normaliseerimine.

4.2 Liitmine ja lahutamine

Kombinatsioonid „00” liitmine või „01” lahutamine, käivitavad liitmise ja lahutamise koodi. Lahutamise puhul inverteeritakse teise liidetava mantissi *mantB* otsekood täiendkoodi. Abimuutujat *minusOneUsed* on kasutatud, et kindlustada ühekordne koodi muutus, sest antud programm vajab täitmiseks enam kui ühe tsükli. Inverteerimisel on kasutatud *signed* tüüpi muutujate korrutamist ja tulemuse vektorisse laadimist (Joonis 12). Vektor *mult* on pikem kui tarvis, aga autor kasutab seda hiljem korrutamise realiseerimisel. Kuna lahutamist saab käsitleda täiendkoodi liitmisena, siis järgnevad sammud saab teha summeerimisega.

Töös kasutatud matemaatiliste tehete juures on kasutatud IEEE numeric_std teeki, mis võimaldab kahendvektoreid käsitleda kui arve.

```
--valik muutjaid ja nende pikkusi, mida inverteerimine kasutab
variable minusOne : STD_LOGIC_VECTOR (13 downto 0):=(others => '1');
variable minusOneUsed : STD_LOGIC:='0';
variable mult : STD_LOGIC_VECTOR (27 downto 0):=(others => '0');

--muudame mantB märki
if minusOneUsed='0' then
   mult := std_logic_vector(signed(mantB)*signed(minusOne));
   --Korrutamine -1 ei pikenda olemas olevat vektorit kohani 27
   mantB := mult(13 downto 0); --  inverteeritud vektori tagastamine
   minusOneUsed:='1';  -- lipp, et oleme juba pööranud
   end if;
else
   minusOneUsed:='0';  -- inverteerimist pole kasutatud
end if;
```

Joonis 12. Muutuja inverteerimine.

Esimese toiminguna (Joonis 13) tuleb kindlaks teha kumma arvu mantiss on suurem. See on kirjas teoreetilises osas 2.1. Hiljem rakendame valem (2) sulgudes olevat mantissi arvutamist kui arvu A astendaja on suurem ja valemi (3) sulgudes olevat osa kui B astendaja on suurem.

```
--Kontrollime, kas astendaja A on suurem B astendajast
if to_integer(signed(powA)) > to_integer(signed(powB)) then
   --Kirjutame selle kohe C astendajaks
   powC:=powA;
else
   powC:=powB;
end if;
--Olenevalt kumb astendaja on suurem tuleb valem kirjutada
if powC=powA then

   powMinus := powB-powA;
   --tshift ütleb kuhu poole liigutada, negatiivne, kuna liigutamine on
   --kirjutatud tagurpidi, nii tuli alati õige vastus
   tshift := to_integer(signed(powMinus)*(-1));
   if powMinus(plen)='1' then
      mantC:=std_logic_vector(shift_right(signed(mantB), tshift))+mantA;
   else
      mantC:=std_logic_vector(shift_left(signed(mantB), tshift))+mantA;
   end if;
else
   powMinus := powA-powB;
   tshift := to_integer(signed(powMinus)*(-1));
   if powMinus(plen)='1' then
      mantC:=std_logic_vector(shift_right(signed(mantA), tshift))+mantB;
   else
      mantC:=std_logic_vector(shift_left(signed(mantA), tshift))+mantB;
   end if;
end if;
```

Joonis 13. Ujupunktarvude liitmise algoritm.

4.3 Korrutamine

Korrutamine toimub kui tehte (*mode)* vektor on „10”. Tehte teostamiseks kasutab autor valemit (5). Korrutamise realiseerib autor kasutades andmetüüpi *signed* ja *numeric_std* teeki (Joonis 14) ning korrutab ja liidab kahendkoodid nagu tavalised märgiga arvud. Tagasi vektoriks konverteerimisel arvestab autor, et tulemuse mantissi vektori lubatud formaat võib olla kuni kaks korrutatava pikkust. Astendajate summa vektori ületäituvus on lubatud. Vektorite valikul on eelnevate tingimustega arvestatud.

```
variable mult : STD_LOGIC_VECTOR (27 downto 0):=(others => '0');
variable powPlus : STD_LOGIC_VECTOR (8 downto 0):=(others => '0');
--korrutamine
   elsif mode = "10" then
    --Tuleb tavaline märgiga arvude korrutamine
    mult:= std_logic_vector(signed(mantA)*signed(mantB));
    --Astendajad liidame, kuna alus on sama
    powPlus:= std_logic_vector(powA(7)&powA) +
std_logic_vector(powB(7)&powB);
```

Joonis 14. Ujupunktarvude korrutamise realiseerimine

Avaldise liikmed on mõlemad normaliseeritud kujul, mis tähendab, et tulemus *mult* vajab enne väljundisse saatmist normaliseerimist. Positiivse tulemuse korral käib tsükkel läbi kõik vektori *mult* elemendid ja otsib esimese „1" (Joonis 15). Kui astendajat saab suurendada arvu võrra, mitu ühikut on tarvis nihutada, siis toimub nihe ja astendaja suureneb. Negatiivse mantissi suurendamine, mille käigus astendajat peab saama vähendada on sarnane positiivsega, erinevus seisneb, et otsime ühe asemel nulli (Joonis 16). Piirid tulevad astendaja formaadi maksimaalsest ja minimaalsest väärtusest.

```
for index in (mlen+mlen+1) downto mlen loop
   if mult(index) > '0'  then --  positiivne korrutis
      --(mlen+mlen-3) Komakohtade arv 23
      if (to_integer(signed(powPlus)) + (index - (mlen+mlen-3)))<=127 then
         --Valime vektori nii, et ette jääb kaks märgi kohta
         mantC:= mult((index+2) downto (index+2-mlen));
         powPlus:=powPlus + (index – (mlen+mlen-3));-- suurendame astendajat
         errorCode<="0000";  --  veakood
      else
         --Positiivset tulemust ei saa vähendada
         errorCode<="1001";
      end if;
   exit;
   end if;
end loop;
```

Joonis 15. Positiivse mantissi vähendamine.

```
for index in (mlen+mlen+1) downto mlen loop
   if mult(index) = '0'  then
      if (to_integer(signed(powPlus)) - (index - (mlen+mlen-3)))>=-65 then
         --Sama mis positiivne, ainult indeks on teises kohas
         mantC:= mult((index+2) downto (index+2-mlen));
         powPlus:=powPlus + (index - (mlen+mlen-3));
      else
         --Negatiivset tulemust ei saa vähendada
         errorCode<="1010";
      end if;
   exit;
   end if;
end loop;
```

Joonis 16. Negatiivse mantissi suurendamine.

4.4 Tehete tulemusena saadud ujupunktarvu formaadi kontroll

Komponendi sees kasutatavad vektorid on ühe ühiku võrra pikemad kui sisend ja väljund, et katta ületäituvus. Ekraanil kuvatakse tehete tulemusel saadud kahendkoodi kümnend kuju, mitte väärtus. Liitmise ja lahutamise järel ei toimud formaadi kontrolli. Piirid on samad, mis korrutamise järgsel kontrollil. Vastavalt märgile toimub astendaja suurendamine või vähendamine. Komponent tagastab muutujad *mant* ja *pow* (Joonis 17).

```
--Piiride kontroll
if to_integer(signed(mantC)) < -3072 or to_integer(signed(mantC)) > 4095 then
   if to_integer(signed(powC))<127 or to_integer(signed(powC))>-96 then
      --Positiivne arv
      if mantC(13)='0' then
         powC := powC+1;  -- Liidame ühe
         mant<=mantC(13 downto 1);
      else
         powC := powC-1; -- Lahutame ühe
         mant<=mantC(13 downto 1);
      end if;
   else
      --Kui vähendamine ei õnnestu anname veakoodi
      errorCode<="1111";
   end if;
else
   mant<=mantC(12 downto 0);
end if;
pow<=powC;
```

Joonis 17. Ületäituvuse likvideerimine.

4.5 Liitmine, lahutamine ja korrutamine 7-segmendilisel indikaatoril

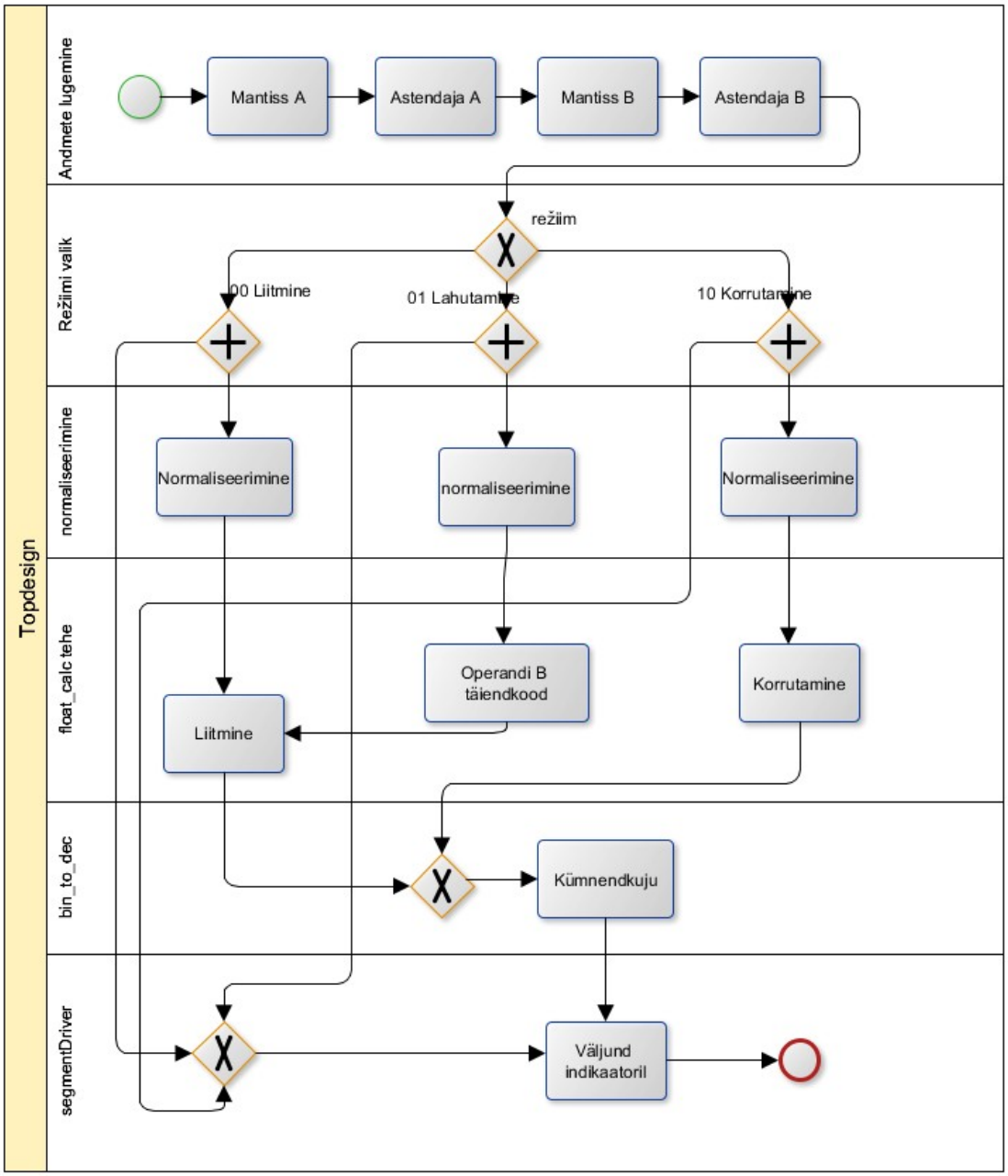
Liitmise puhul kuvatakse ekraanil tulemus ja vastavalt mantiss „M3" või astendaja „P3" (Joonis 18). Kuna astendaja formaat on lühem, lisatakse talle ette märgi väärtused. Märk arvu ees ei muuda tema väärtust.

```
if btnReg = "10000" and btnDeBnc ="00000" then
   if SW(14 downto 13)="00" then
      numbers(31 downto 24):="10100011"; --M3
      onscreen<=mant;
   elsif SW(14 downto 13)="01" then
      numbers(31 downto 24):="10110011"; --P3
      onscreen(12 downto 7)<=pow(7)&pow(7)&pow(7)&pow(7)&pow(7)&pow(7);
      onscreen(6 downto 0)<=pow(6 downto 0);
   end if;
  --Veakood ekraaanile
   if errorCode(3)='1' then
      numbers(31 downto 0):=(others=>'1');
      onscreen<=(others=>'0');
      onscreen(3 downto 0 )<=errorCode;
      end if;
   end if;
```

Joonis 18. Liitmise tulemus 7-segmendi indikaatorile.

Sarnaselt liitmise kuvamisele, toimub ka lahutamise ja korrutamise kuvamine. Muutub vaid registreeritud nupuvajutus. Lahutamisel on ekraanil „M4" ja „P4". Korrutamisel „M5" ja „P5".

Autor lisas tööle lihtsustatud ülevaatliku skeemi, kuidas kasutaja poolt sisestatud andmed jõuavad indikaatorile (Joonis 19).



Joonis 19. Programmi käitumise skeem.

5 Testimine

Testida saab kui genereerida VHDL koodist bitikood ja saata see seadmesse, eelneb süntees ja rakendamine. Antud protsess võib võtta ligi 10 minutit, mis sagedaste muutuste juures teevad testimise keeruliseks. Üks võimalus on luua simulatsioon. Simuleerida saab hierarhia tipus olevat skeemi eraldi või luua testpink mingi kindla komponendi testiks. Testpink ei erine tavalisest siduvast VHDL failist. Oma lõputöö katsetuste käigus olen kasutanud mõlemat meetodit.

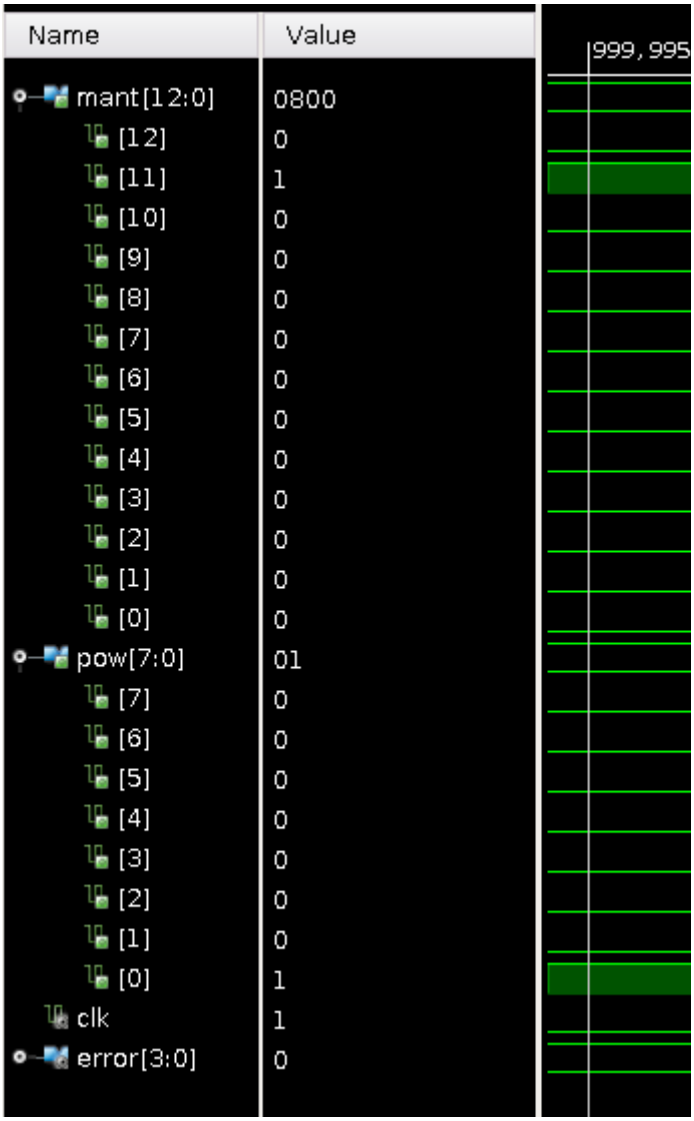
5.1 Testpink ujukoma arvutuste testimiseks

Joonisel 20 oleva testiga soovib autor testida, kas korrutamise tulemus on ikka üks. Lihtne test võimaldab kontrollida korrutamist ja arvu formaadi taastamist pärast kontrolli, kuna koodis pole sees eraldi tingumust, et kui arvud on võrdsed, siis käsitle neid teisiti.

```
--Kuidas siduda ja anda väärtused
test_float_calc: float_calc PORT MAP(
   clk =>CLK,
   errorCode=>error,
   mantA_in => "0100000000000", --0.5
   mantB_in => "0100000000000", --0,5
   powA_in => "00000001", -- 1
   powB_in => "00000001", -- 1
   mode => "10",
   mant => mant,
   pow =>pow
);
```

Joonis 20. Testpink ujukomatehetele.

Testide tulemust saab visuaalselt uurida. Antud juhul test õnnestus, sest autor parandas programmis leiduvad loogika vead varasemate testide järel. Esialgne realiseerimise loogika nihutamisel ei arvestanud komakohtade arvuga.

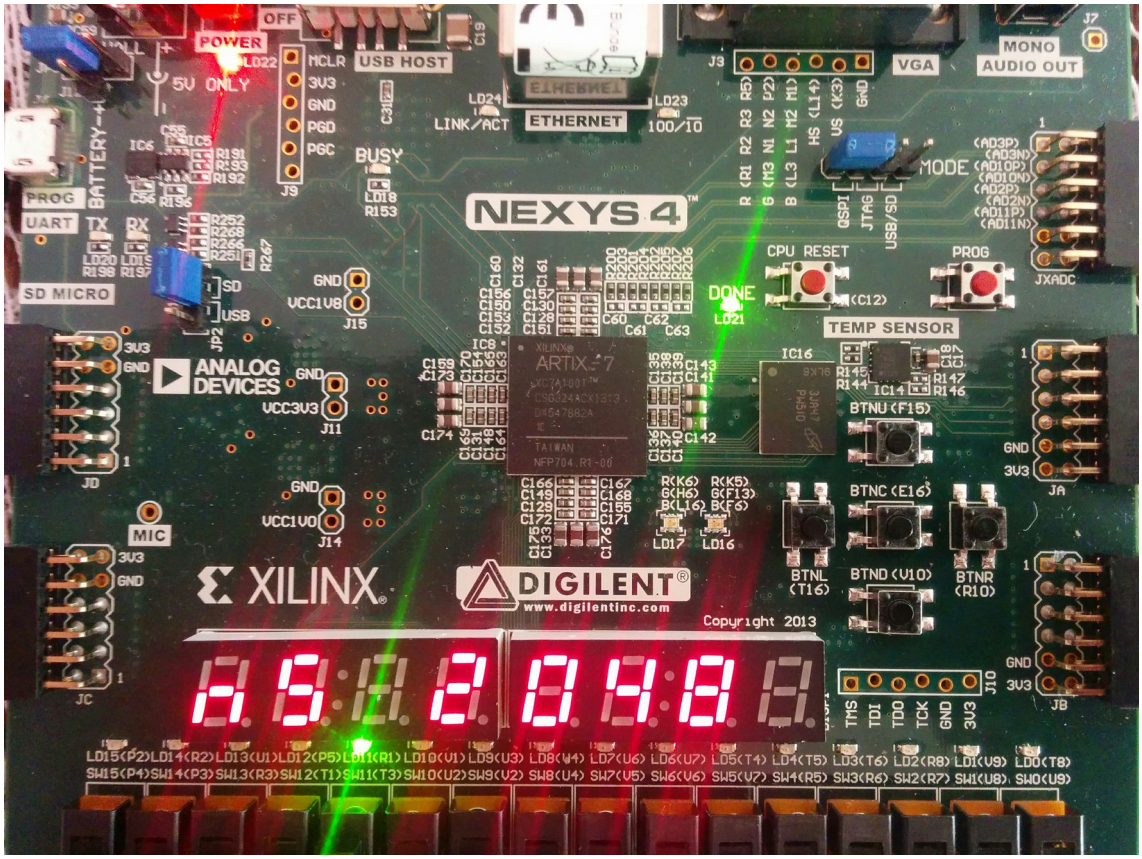


Joonis 21. Testi tulemus.

5.2 Testimine seadmel

Peatükis 5.1 kasutusel olnud ja sarnaste testide sooritamisel seadmel Digilent Nexys 4, selgus, et indikaatoril indeksiga kolm esineb defekt või on programmi loogikas viga, mis saadab kattuvad signaalid väljundisse. Paraku ei suutnud tuvastada programmi loogikas ühtegi viga, mille tulemusel indikaator 3 vilgub. Indikaator ei vilgu alati vaid siis kui väärtuse kuvamiseks esimene number on 2. Lahendusena nihutas autor väljundi

ühe ühiku võrra vasakule (Joonis 22). Rohkem erinevusi testpingil saadud väärtuste ja seadmel testitud tulemuste vahel polnud.



Joonis 22. Korrutamise tulemuse mantiss seadmel.

5.3 Komponendi simuleerimine kasutades Tcl käske

Arenduskeskkonnas Xilinx Vivado on võimalik rutiinsed tegevused automatiseerida, kasutades selleks Tcl skripti. Nii saab simulatsioonis käsitsi sisestavaid väärtusi skriptiga lisada ja kontrollida ekraanil, kas tulemus tuleb õige.

Käesolev testmeetod on töökindlam võrreldes ühe komponendi käitumise testpingiga, kuna võimaldab näha tulemust, mida kuvatakse indikaatoritele. Väärtused määramata (U) ja tundmatu (X) simulatsioonitulemustes tekitavad küsimusi, kas programm töötab ikka nii nagu vaja. Joonis 23 selgitab kuidas seada lülitile vektor nii, et märgi bitt on positiivne, sisse loetakse esimene mantiss, väärtusega kolm. Surunupu vektori

registreerimiseks tuleb Tcl skriptiga tekitada registreeritud muutus. Testide ajal vähendas autor *debounceri* nupuvajutuse registreerimise tsüklite arvu 2^{16-} lt kaheksani.

```
#Sisend
#Mant A 11 lüliti nupud 14 13 nulli ja nupuvajutused väikse vahega
add_force {/TopDesign/SW} -radix bin {0000000000000011 0ns}
#takt toimub muutub iga 10 nanosekundi järel, kahekümne taktiga peaks kõik
#tehtud saama
run 200ns
add_force {/TopDesign/BTN} -radix bin {00000 0ns}
run 200ns
add_force {/TopDesign/BTN} -radix bin {00001 0ns}
run 200ns
add_force {/TopDesign/BTN} -radix bin {00000 0ns}
run 200ns
```

Joonis 23. Mantiss A sisestamine.

6 Versioonihaldus

Oma töös kasutab autor versioonihaldustarkvara Git, mis võimaldab talletada projekti erinevad etapid. Arenduskeskkond Xilinx Vivado võimaldab luua projekti kasutades ainult Tcl skripti ja käsurida, mis lubab näiteks saale- ja rakendusfailid varukoopiast välja jätta. Täpse skripti sisu ja projekti struktuur on lahti seletatud juhendis (Lisa 2 – Xilinx Vivado projekti loomine kasutades skripti). Xilinx Vivado ei võimalda iseseisvalt VHDL programmi erinevaid versioone tekitada. Võttes projektist vaid failid, mis on nõutud toimiva versiooni loomiseks, ja seome need Git projektiga, saab luua koosluse, kus töötab versioonihaldus.

Programmi Git seadete hulka kuulub fail *.gitignore*. Failis kirjeldatakse kaustad ja failid, mida projekti ei kaasata. Joonis 24 selgitab, millised failid ja kaustad võib Xilinx Vivado projektist välja jätta, nii et skripti abil saab luua töötava versiooni.

```
# Kaustad ja failid, mille võib välja jätta. Kuna minu projekti nimi on FPU,
# siis kõik kaustad nimega FPU jäävad välja: sim, impl, runs, hw, cache
proj/FPU.*
proj/*.log
proj/*.jou
```

Joonis 24. Ignoreeritavate failide ja kaustade muster.

7 Kokkuvõte

Lõputöö teostamise käigus tutvus autor ujupunktarvude teoreetilise baasiga. Üheks töö ülesandeks on koostada ujupunktarvude tehete algoritmid ja need realiseerida VHDL keeles. Autor valis töö jaoks liitmise, lahutamise ja korrutamise. Soovitusliku formaadi pakkus välja juhendaja. Ujukomaehete teoreetilise materjali põhipunktid on selgitatud Peatükis 2.

Töö realiseerimiseks sai autor kasutada seadet Digilent Nexys 4. Kuna autori omanduses puudusid teegid, mis võimaldavad kasutada väljundina 7-segmendi indikaatorit, siis lisandus algsetele ülesannetele vastavate komponentide loomine ja testimine.

Arenduskeskkonnas Xilinx Vivado kasutas autor rohkesti Tcl skripti võimalusi. Peamiselt projekti loomisel ja testimisel. Autori töökogemus tarkvaraarendajana võimaldas märgata Digilent poolt pakutud näidisprojektide seost versioonihalduse võimalustega. Bakalaureuse töö juurde kuulub detailne juhend kuidas Xilinx Vivado abil luua Digilent Nexys 4 tarbeks töötav bitikood (Lisa 1 – Arenduskeskkonna Xilinx Vivado kasutusjuhend).

Ujukomatehte algoritmid on kirjeldatud ja realiseeritud VHDL keeles. Operandide sisestamisel on lubatud normaliseerimata kuju ning normaliseerimist on põhjalikult käsitletud Peatükis 4.1. Samuti toimub väljundi normaliseerimine, sest näiteks korrutamisel võib tulemuseks saadav mantiss olla kaks korda pikema formaadiga kui operandid.

Sisendite, väljundite ja ujukomatehete programmid on testitud vastavate käitumise simulatsioonidega ning füüsilisel platvormil. Testide käigus avastatud vead on parandatud.

Kasutatud kirjandus

[1] Arvutite aritmeetika ja loogika IAY0140 harjutustundide materjal [WWW], http://www.diskmat.ee/aaritm.pdf (11.04.1016)

[2] Digilent Nexys 4™ FPGA Board Reference Manual [WWW] https://reference.digilentinc.com/_media/nexys/nexys4/nexys4_rm.pdf (15.04.2016)

[3] Github Projekt FPU [WWW] https://github.com/rometkoiv/FPU(12.05.2016)

[4] Lensen, H., Kruus, M. Diskreetne matemaatika, Tallinn: Tallinna Tehnikaülikooli Kirjastus, 2012.

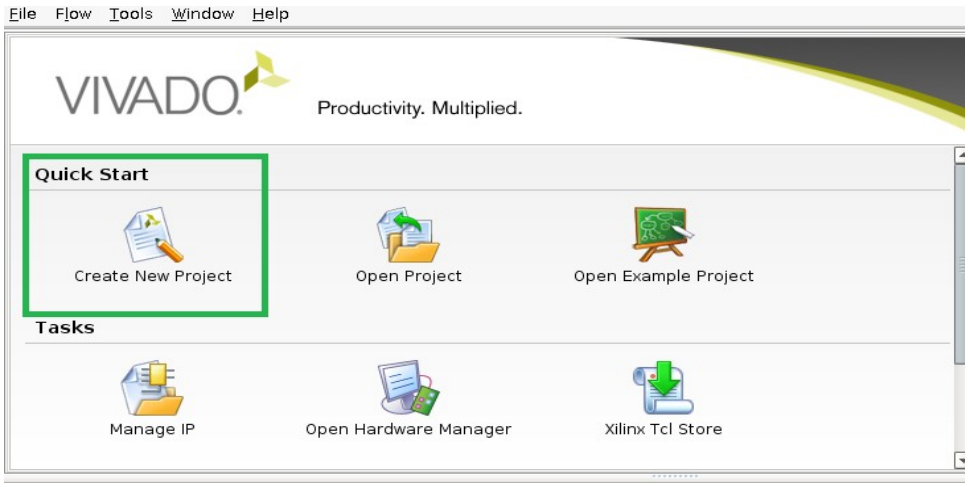
[5] The Official Digilent Github, Nexys4, project GPIO [WWW] https://github.com/Digilent/Nexys4/blob/master/Projects/GPIO/src/hdl/debouncer.vhd (01.05.2016)

[6] The VHDL Cookbook, First Edition, Peter J. Ashender, 1990 [WWW] http://www.ics.uci.edu/~alexv/154/VHDL-Cookbook.pdf (16.04.2016)

[7] Xilinx 7 Series FPGAs Overview, DS180 Product Specfication. [WWW] http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (11.05.2016)

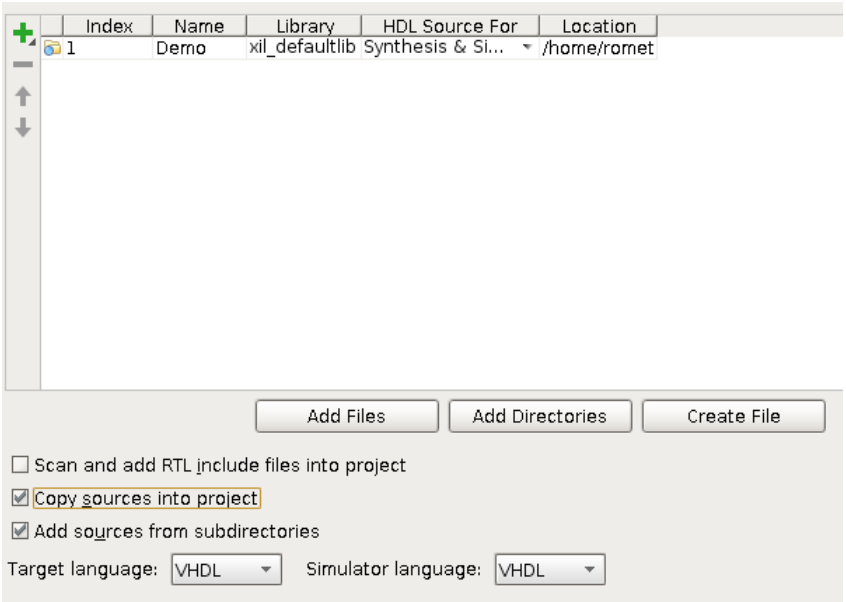
Lisa 1 – Arenduskeskkonna Xilinx Vivado kasutusjuhend

Uue projektiga alustamiseks on soovitatav kasutada loomise viisardit (Joonis 25).



Joonis 25. Uue projekti loomise nupp

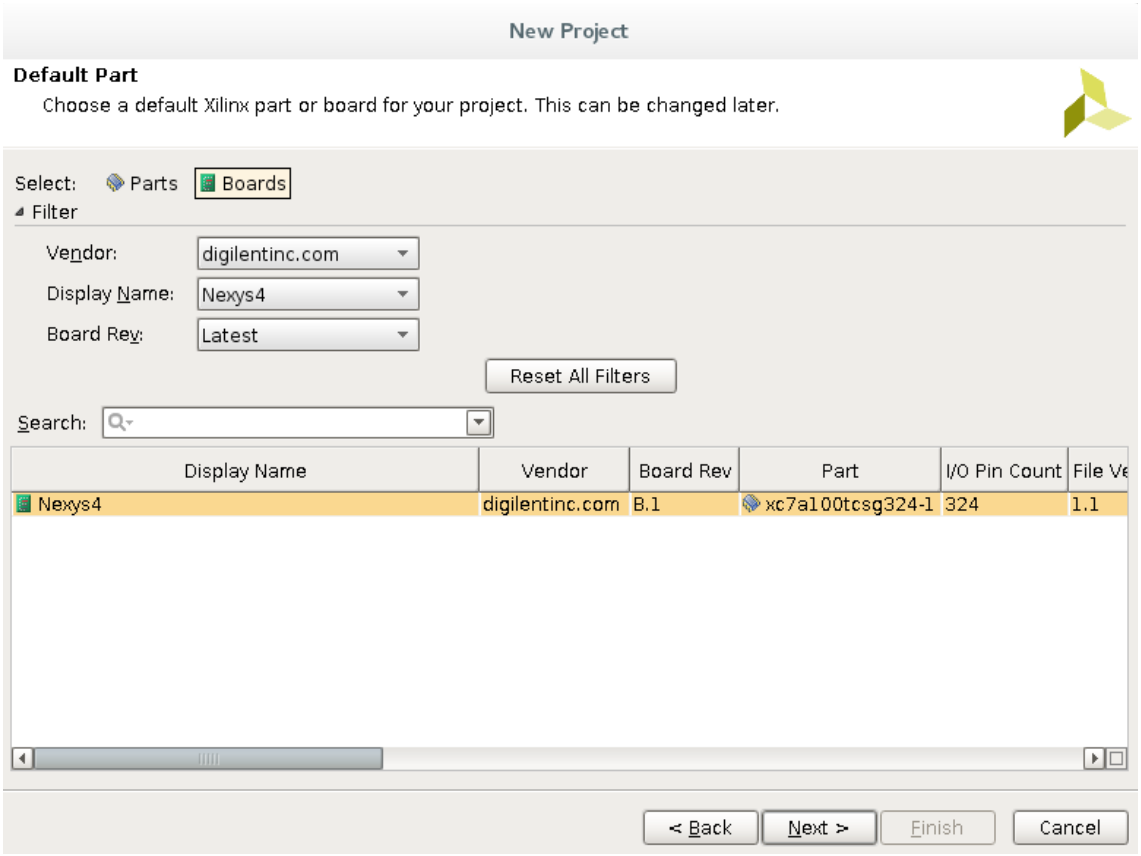
Esimesest akna seaded võib jätta samaks. VHDL projekti keele puhul tuleb määrata *Target language* ja *Simulation language* (Joonis 26). Soovitatav on kohe luua vähemalt üks VHDL fail või linkida olemas olevad.



Joonis 26. Programmi keele seadmine.

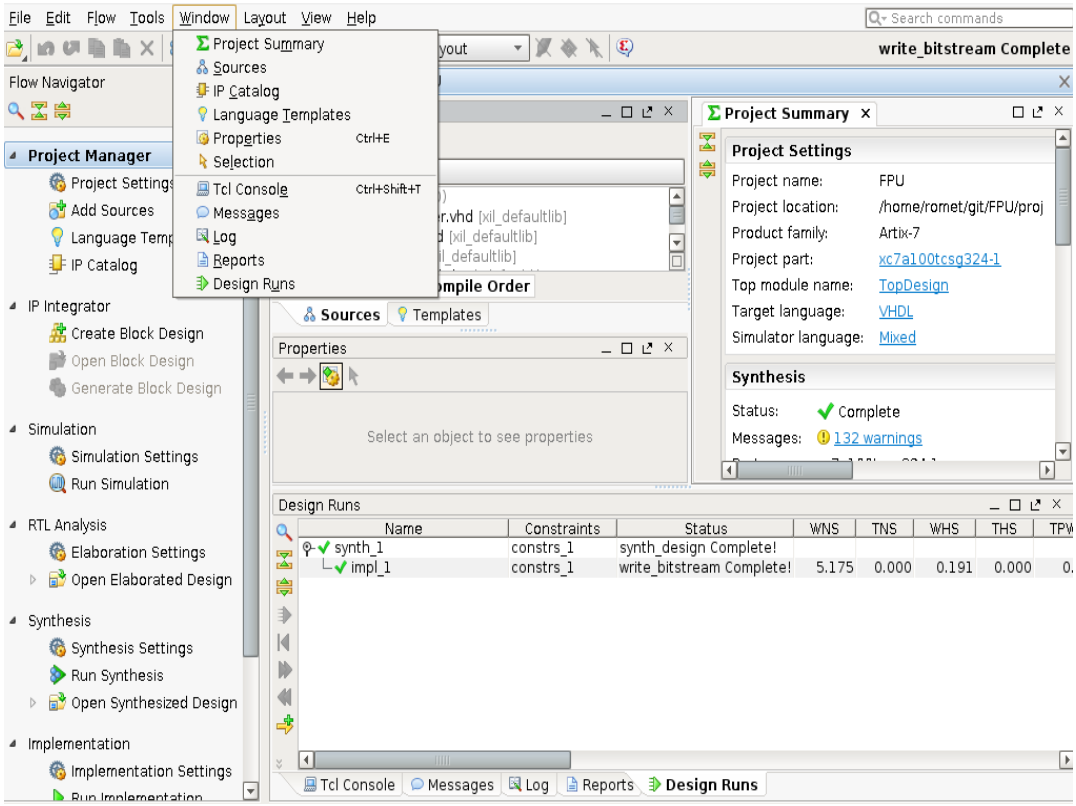
Valida plaat (*board*). Seadme tootjaks (*vendor*) valida *digilentinc.com*. Edasi valida seade (*Display name*) Nexys 4 viimane versioon (Joonis 27). Kui programm ei paku näites toodud plaati, tuleb tootja kodulehelt alla laadid vastavad draiverid.

Järgnevad valikud võib jätta vaikimisi pakutuks ning käsuga *finish* luua projekt.



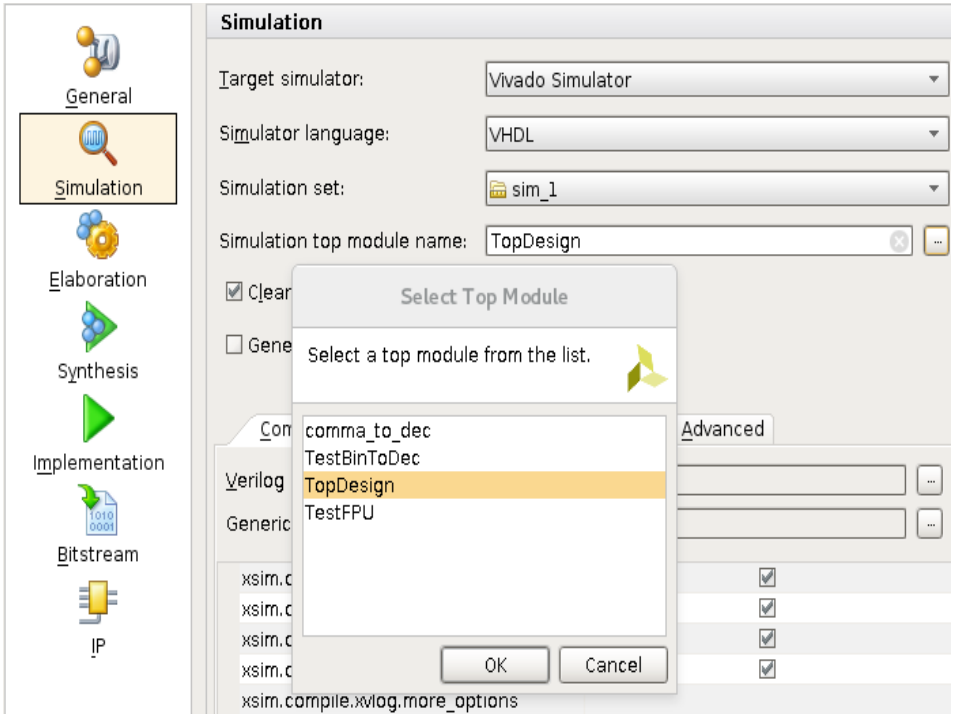
Joonis 27. Projekti seadme valimine.

Väiksema ekraanilahutuse korral võib juhtuda, et projekti vaates pole võimalik valida ühegi alamakna sisu. Soovitatav on sulgeda alamaknad mida ei kasuta ja vajadusel need *Window* menüüst taasavada (Joonis 28). Kui kirjutada programmi, on tarvis näha akna *Source* sisu ja redigeeritavat faili. Muud aknad võib sulgeda.



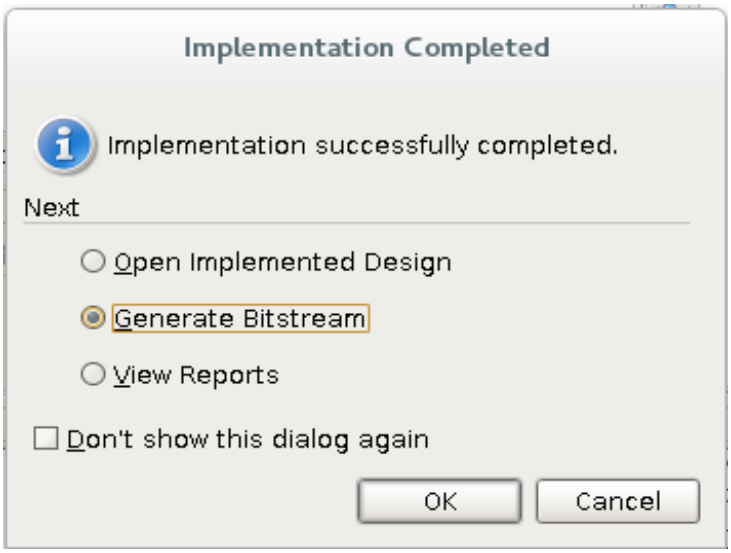
Joonis 28. Projekti vaikimisi vaade ja alamaknad.

Simulatsiooni läbiviimiseks on tarvis luua seaded (*Simulation set*). Valida disain, mida simuleeritakse, tegemist ei pea olema pealmise disaini failiga.



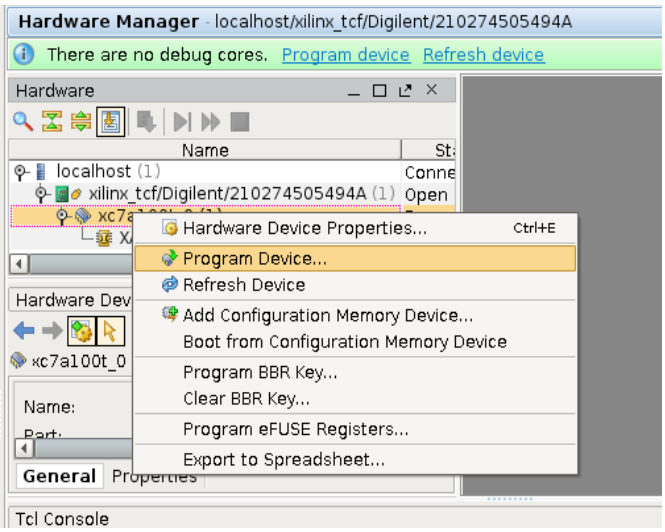
Joonis 29. Vivado projekti simuleerimise sätted.

Pärast edukat simuleerimist võib genereerida bitijada. Soovitav on alustada rohelisest kolmnurgast *Generate implementation* (asub menüü *Layout* all). Antud toiming kontrollib kas süntees on ajakohane ja vajadusel loob uue. Järgneb implementatsioon, mille lõpus küsitakse, kas loome bitijada (Joonis 30).



Joonis 30. Implementatsiooni alustamine.

Mõnikord võib juhtuda, et projekti aknast kaob riistvarahaldur (*Hardware Manager*), siis saab selle avada *Flow* menüüst. Riistvarahalduri abil saab seadet programmeerida. Kõigepealt tuleb avada parema hiireklahviga seade meetodiga *auto connect* või valida seade käsitsi. Programmeerimiseks parem hiireklikk ja *program device* (Joonis 31).

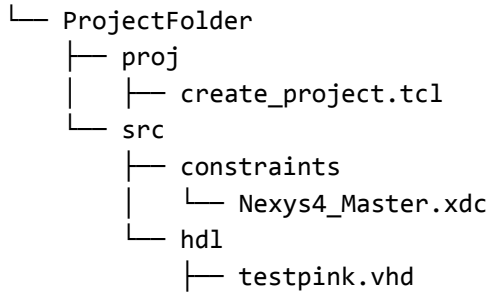


Joonis 31. Seadme programmeerimine.

Lisa 2 – Xilinx Vivado projekti loomine kasutades skripti

Antud juhend käsitleb vaid Tcl skriptiga projekti loomist. Kasutades selleks näite projekte Digilent Nexys4 *githubis* (https://github.com/Digilent/Nexys4, edaspidi *github*).

Kõigepealt on tarvis luua projekti kaust (kasutame nime *ProjectFolder*, mis asub kodukaustas), milles peavad olema järgmised kaustad ja failid (Joonis 32). Sama struktuur on kasutuses ka Digilent projektidel. Lisaks kaustadele tuleb all laadida piirangute fail. Soovitan kasutada projekti GPIO src/constraints/Nexys4_Master.xdc faili. Projekti loomise skripti soovitan alla laadida samast projektist. proj/create_project.tcl



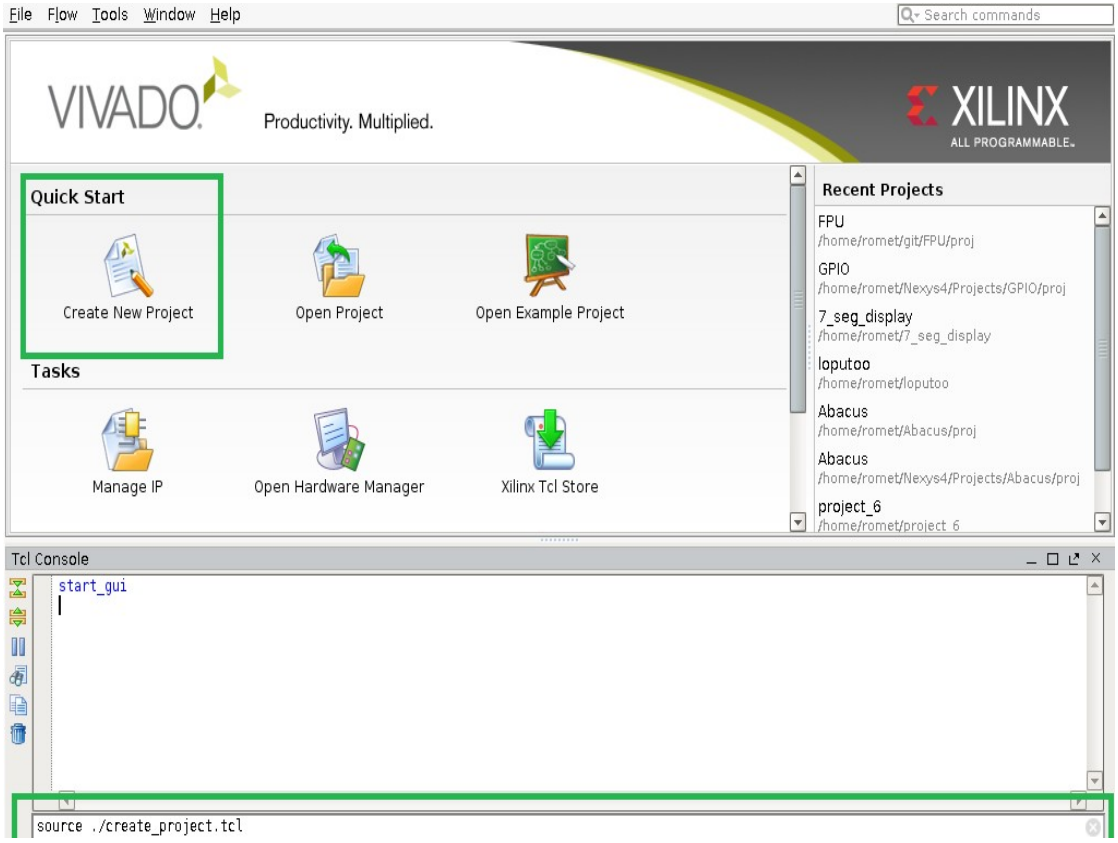
Joonis 32. Kaustade struktuur.

Piirangute faili tuleb muuta vastavalt kasutatavale vhdl koodile. Projekti loomise faili võib samaks jätta. Soovitan muuta projekti nime ja veenduda, et seade on ikka sama, millele projekt on loodud(Joonis 33).

```
#Projekti nimi
set proj_name "MinuProjekt"
# Seadme number, veenduda, et on sama mis reaalsel seadmel
set part_num "xc7a100tcsg324-1"
```

Joonis 33. Muutused projekti loomise failis.

Kui on juba valmis VHDL faile, näiteks testpink.vhd, siis need võib paigutada src/hdl kausta. Edasi avada Vivado ja viia kursor TCL Console käsureale (Joonis 34) ning teostada järgnevad toimingud (Joonis 35).



Joonis 34. Käsurea asukoht Xilinx Vivados.

Kui projekti kooseisus on korrektsed VHDL failid, võib kohe peale projekti loomist genereerida töötava bitijada, viies läbi sünteesi ja implementatsiooni.

```
#teha kindlaks kus kasutas asub hetkel konsool
pwd
#Kui tegu pole projektiga, minna antud kausta ja kohe tema alamkausta proj
cd ~/ProjectFolder/proj/
#Käivitada projekti loomise skript
source ./create_project.tcl
```

Joonis 35. Projekti loomine käsurealt.

Plaadile saab kirjutada kui kasutada Riistvarahaldurit (*Hardware Manager*). Xilinx Vivado juhend (Lisa 1 – Arenduskeskkonna Xilinx Vivado kasutusjuhend).

Kui on soov kasutada lõputöö aluseks olevat programmi [3] , saab selle masinasse laadida kasutades git käsku *clone* (Joonis 36).

```
git clone --branch V1.1 https://github.com/rometkoiv/FPU.git
#Käivitada Vivado. Liikuda kasuta FPU/proj ja luua projekt
source ./create_project.tcl
```

Joonis 36. Kloonitud programmist projekti loomine