Tampereen yliopisto

Samu Prusi

# AVOIMEN LÄHDEKOODIN LIUKULUKU-YKSIKKÖTOTEUTUKSET FPGA:LLE

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Matti Haavisto
Toukokuu 2020

# TIIVISTELMÄ

Samu Prusi: Avoimen lähdekoodin liukulukuyksikkötoteutukset FPGA:lle
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaatin tutkinto-ohjelma
Toukokuu 2020

---

Liukuluvut ovat merkittävässä asemassa tietokoneella tehtävässä laskentatyössä, sillä kokonaisluvuilla ei kyetä suorittamaan kaikkia laskuja. Liukuluvut tarjoavat reaaliluvuille laajan lukuskaalan, jolloin voidaan esittää suurten lukuarvojen lisäksi pienet lukuarvot riittävällä tarkkuudella. Liukuluvuilla tehtävää aritmetiikkaa suoritetaan liukulukuyksikössä.

Tässä työssä etsitään avoimen lähdekoodin liukulukuyksikkötoteutuksia ja implementoidaan niitä FPGA-alustoille. Implementaatiotulosten perusteella tarkastellaan yksiköiden resurssien- ja tehonkulutusta sekä ajoitustietoja. Työhön valittiin kolme laitteistonkuvauskielellä toteutettua yksikköä sekä yksi korkeammalla abstraktiotasolla kuvattu yksikkö.

Ensimmäisessä mittausosiossa vertailtiin laitteistonkuvauskielitoteutuksia implementoimalla niitä samalle alustalle. Mittaustulosten perusteella todettiin yksiköiden resurssien- ja tehonkulutuksen lisääntyvän yksiköiden tarkkuuden ja kellotaajuuden kasvaessa.

Toisessa mittausosiossa tarkasteltiin miten mitattavat parametrit muuttuvat kohdealustan vaihtuessa. Mittaustulosten perusteella todettiin yksiköiden ajoitusten parantuvan kohdealustan vaihtuessa tehokkaampaan. Vastaavasti huomattiin ajoitusten heikkenevän ja voivan jopa epäonnistua vaatimattomammalla kohdealustalla. Jokaisen laitteistonkuvauskielitoteutuksen tehonkulutus kasvoi siirryttäessä tehokkaammalle alustalle. Mittauksissa nousi esille, että implementoitaessa yksiköitä tehokkaammalle alustalle alustan tehonkulutus nousi itse toteutuksen tehonkulutusta merkittävämmäksi. Tällöin tehokkaamman alustan valinta liukulukuyksikölle ei välttämättä ole enää perusteltua, jos yksikkö saadaan implementoitua ja ajoitettua onnistuneesti myös vaatimattomammalla alustalla.

Kolmannessa mittausosiossa vertailtiin laitteistonkuvauskielellä ja korkeamman tason kuvauksella toteutettuja yksiköitä. Mittaustulosten perusteella korkeamman tason toteutus vaatii laitteistonkuvauskielitoteutukseen verrattuna enemmän resursseja ja kuluttaa enemmän tehoa. Tällöin todettiin laitteistonkuvauskielellä toteutetun rekisteritason kuvauksen olevan vielä tehokkaampi vaihtoehto liukulukuyksiköiden toteutukseen. Korkeamman tason toteutuksen todettiin kuitenkin vaativan vielä tarkempaa tutkimusta, sillä sen tarjoamista operaatioista mitattiin ainoastaan murto-osaa. Korkeamman tason kuvauksen edut saattavatkin tulla ilmi vasta monimutkaisempia aritmeettisia operaatioita toteutettaessa.

Avainsanat: Liukulukuyksikkö, FPU, FPGA

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

# SISÄLLYSLUETTELO

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| DSP | engl. Digital Signal Processing, digitaalinen signaalinkäsittely |
| FloPoCo | engl. Floating Point Core |
| FPGA | engl. Field Programmable Gate Array, uudelleenohjelmoitava logiikkapiiri |
| FPU | engl. Floating-Point Unit, liukulukuyksikkö |
| GPU | engl. Graphics Processing Unit, grafiikkaprosessori |
| IEEE | engl. Institute of Electrical and Electronics Engineers, kansainvälinen tekniikan alan järjestö |
| LSB | engl. Least Significant Bit, vähiten merkitsevä bitti |
| MSB | engl. Most Significant Bit, eniten merkitsevä bitti |
| NaN | engl. Not a Number, määrittelemätön numeroarvo |
| VHDL | engl. Very high speed integrated circuit Hardware Description Language, laitteistonkuvauskieli |
| WHS | engl. Worst Hold Slack |
| WNS | engl, Worst Negative Slack |

| | |
|---|---|
| *bias* | eksponentin biasointiarvo |
| *e* | eksponentti |
| *m* | mantissa |
| *p* | liukuluvun tarkkuus |
| *s* | etumerkki |
| *t* | mantissan bittimäärä |
| *w* | eksponentin bittimäärä |
| *β* | kantaluku |

# 1. JOHDANTO

Liukuluvut ovat merkittävässä asemassa tietokoneella tehtävässä laskentatyössä. Vain murto-osa laskuista voidaan esittää ainoastaan kokonaisluvuilla, jolloin tarvitaan keino tietokoneelle reaalilukujen esittämiseen. Ratkaisun tähän ongelmaan tarjoavatkin nimenomaan liukuluvut. Liukuluvuilla voidaan esittää eri lukuarvoja laajalla skaalalla, jolloin saavutetaan samanaikaisesti suuret lukuarvot sekä merkittävä tarkkuus pienillä lukuarvoilla. Moderneissa tietokoneissa prosessoreihin onkin integroitu yksi tai useampi liukulukuyksikkö, jotka nimensä mukaisesti hoitavat tarvittavan liukulukulaskennan.

Liukulukujen käsittelyn tärkeyden vuoksi liukulukuyksiköt ovat jatkuvasti ajankohtainen aihe tietokonetekniikan tutkijoiden sekä kehittäjien keskuudessa. Nykyään tärkeäksi tutkimusalustaksi tietokonetekniikan alalla ovat nousseet erilaiset FPGA-alustat (engl. Field Programmable Gate Array), sillä ne tarjoavat halvan hinnan, helpon integraation ja suorituskyvyn yhdistelmällä oivan alustan myös liukulukuyksiköiden kehitykseen. Tämän työn tarkoituksena on selvittää, millaisia liukulukuyksiköiden FPGA-toteutuksia on saatavilla. Työssä etsitään erilaisia avoimen lähdekoodin liukulukuyksiköitä, joita implementoidaan FPGA-alustoille logiikkasynteesityökalulla. Implementaation jälkeen tarkastellaan yksiköiden resurssien- ja tehonkulutusta synteesityökalun tarjoamien tietojen perusteella. Lisäksi tarkastellaan kuinka kohdealustan vaihto vaikuttaa edellä mainittuihin parametreihin sekä yksiköiden ajoituksiin. Lopuksi vielä vertaillaan eri abstraktiotasoilla kuvattujen yksiköiden parametreja implementaatiotulosten avulla.

Työn toisessa luvussa perehdytään liukulukujen perusteisiin sekä tapaan, jolla ne esitetään tietokoneissa. Kolmannessa luvussa tarkastellaan liukulukuyksiköiden rakennetta. Neljännessä luvussa tutustutaan valittuihin liukulukuyksiköihin ja työn mittausjärjestelyihin. Viidennessä luvussa kootaan mittauksista saadut tulokset. Kuudennessa luvussa kootaan työstä saadut tulokset yhteen ja esitetään niistä johtopäätökset.

# 2. LIUKULUVUT

*Liukuluvut* ovat tietokoneen tapa esittää reaalilukuja. Liukulukuja määriteltäessä on otettava huomioon niille asetetut vaatimukset, jotta liukulukujen edut esimerkiksi kiintolukuihin verrattuna tulevat ilmi. Liukulukujen vaatimuksiin voidaan sisällyttää suuren lukuskaalan lisäksi esimerkiksi nopeus, tarkkuus ja helppo toteutus. Käytettävä liukulukujen esitystapa muodostuukin näiden vaatimusten kompromissina. Esimerkiksi vaikka liukulukulaskennan tulos halutaan mahdollisimman nopeasti, ei nopeutta voida kuitenkaan parantaa tarkkuuden kustannuksella. Jos taas esitystapa on liian monimutkainen, sen toteuttamisen hyödyt voivat jäädä pieniksi. [1]

## 2.1 Liukulukujen esitystapa

Nykyisin käytetyt liukulukujen esitystavat pohjautuvat kansainvälisen tekniikan alan järjestö IEEE:n (engl. Institute of Electrical and Electronics Engineers) määrittelemään standardiin. Tämä standardi määrittelee liukulukujen eri esitystavat sekä niiden käyttäytymisen eri tilanteissa. Ensimmäinen liukulukustandardi IEEE 754-1985 [2] otettiin käyttöön vuonna 1985 ja sen seuraaja IEEE 754-1987 [3] vuonna 1987. Standardi otettiin uudelleen tarkasteluun yli 20 vuotta myöhemmin ja uusi paranneltu standardi IEEE 754-2008 [4] otettiin käyttöön elokuussa 2008.

IEEE 754 määrittelee, että liukulukujen esitystapa on yksikäsitteinen, jolloin jokainen luku voidaan esittää vain yhdellä tavalla. Standardin mukaan liukuluku $x$ voidaan esittää neljän kokonaisluvun avulla muodossa

$$x = (-1)^s \cdot m \cdot \beta^e, \qquad (1)$$

jossa ***s*** on luvun etumerkki, ***m*** liukuluvun mantissa, $\beta$ on liukulukujärjestelmän kantaluku ja ***e*** luvun eksponentti. Määritellään lisäksi, että $s \in \{0,1\}$ ja $m \geq 0$. [5] Koska ***s*** voi saada vain arvot 0 tai 1, voidaan siis luvun etumerkki määrittää yhden bitin avulla. Kaavassa (1) esiintyvästä lausekkeesta $(-1)^s$ seuraa, että kun ***s*** on 0, luku on positiivinen, ja vastaavasti arvolla 1 luku on negatiivinen. Luvun mantissa ***m*** kuvaa sen merkitseviä numeroita [5]. Mantissa on määritelty normalisoiduksi, mikä tarkoittaa, että sen desimaalierottimen vasemmalla puolella on ainoastaan yksi numero. Lisäksi tarkennetaan, että tämän numeron ollessa 0 luku on ei-normalisoitu. [1] IEEE:n uusimmassa liukulukustandardissa on määritelty luvut 2 tai 10 mahdollisiksi kantaluvuiksi liukulukuaritmetiikkaa varten [4]. Koska tietokoneet käyttävät binääristä esitystapaa, toimii pääasiassa myös liukulukujen

kantalukuna 2 [6]. Liukuluvun esitykseen kuuluu myös sen tarkkuus *p*, joka kuvaa luvun merkitsevien numeroiden määrää. Tällöin luvun mantissa sisältää enintään tarkkuuden rajaaman määrän numeroita kantaluvulla $\beta$.

IEEE 754 määrittelee lisäksi vielä kaavassa (1) esiintyvälle eksponentille *e* ala- ja ylärajat siten, että $e_{min} < e < e_{max}$. Eksponentin rajat määritellään siten, että $e_{min} = 1 - e_{max}$ [4] ja tästä johtuen käytännön sovelluksissa aina myös $e_{min} < 0 < e_{max}$ [1]. Nyt eksponentin rajojen sekä tarkkuuden *p* avulla voidaan määrittää liukuluvun pienimmäksi esitettäväksi arvoksi $\beta^{e_{min}-1}$. Vastaavasti suurimmaksi mahdolliseksi arvoksi voidaan määrittää $\beta^{e_{max}}(1 - \beta^{-p})$. [1]

Kuten edellä todettiin, liukuluvun lukuarvojen skaalan määräävät sen eksponentin rajaarvot sekä sen tarkkuus *p*. Nämä arvot määräytyvät liukuluvun sananpituuden eli bittimäärän mukaan, ja IEEE:n alkuperäinen standardi 754-1985 määrittelee binääriselle liukuluvulle kaksi eri peruspituutta. Nämä tarkkuudet ovat perustarkkuus, jonka pituus on 32 bittiä, ja kaksoistarkkuus, jonka pituus on 64 bittiä. [2] Uusittu vuoden 2008 standardi määrittelee uudelleen liukuluvuille neljä eri tarkkuutta. Nämä tarkkuudet ovat binary16, binary32, binary64 ja binary128. [4] Jokaisen tarkkuuden perässä oleva numero kertoo niissä käytetyn sananpituuden. 32- ja 64-bittiset uudet tarkkuudet vastaavat alkuperäisiä perus- ja kaksoistarkkuuksia, ja standardiin on lisätty uusi nelinkertainen tarkkuus binary128 [4]. Taulukkoon 1 on koottu nykyisin käytettäviä tarkkuuksia vastaavat tärkeimmät parametrit.

Taulukko 1. *Liukulukujen tarkkuudet sekä niitä vastaavat parametrit [4]*

| **Parametri** | **binary16** | **binary32** | **binary64** | **binary128** |
|---|---|---|---|---|
| sananpituus (bittiä) | 16 | 32 | 64 | 128 |
| *p* | 11 | 24 | 53 | 113 |
| $e_{max}$ | +15 | +127 | +1023 | +16383 |
| $e_{min}$ | -14 | -126 | -1022 | -16382 |
| *bias* | 15 | 127 | 1023 | 16383 |
| *w* (bittiä) | 5 | 8 | 11 | 15 |
| *t* (bittiä) | 10 | 23 | 52 | 112 |

Standardin määrittelemien parametrien perusteella voidaan kullekin tarkkuudelle määrittää pienimmät ja suurimmat esitettävät lukuarvot edellä kuvatulla tavalla. Taulukossa 1 on myös esitetty kunkin tarkkuuden käyttämät bittimäärät. Taulukossa esiintyvä *w* kuvaa eksponentin esittämiseen varattua bittimäärää. Parametri *t* taas kuvaa mantissan esittä-

miseen käytettävää bittimäärää. [4] Taulukossa esiintyvä *bias* kuvaa eksponentin esittämiseen käytettävää biasointiarvoa. Bittimäärällä *w* voidaan esittää etumerkittömät kokonaisluvut nollasta arvoon $2^w - 1$ . Jotta biteillä voidaan esittää myös eksponentin negatiivisia arvoja, vähennetään *w*:n esittämästä arvosta biasarvo, jolloin saadaan selville todellinen eksponentin arvo. [1] Kuvassa 1 on havainnollistettu bittien järjestys liukuluvussa.


***Kuva 1.*** *Liukuluvun bittien järjestys [4]*

Kuvassa esiintyvä MSB (engl. Most Significant Bit*)* kuvaa bittijonon eniten merkitsevää bittiä. Vastaavasti taas LSB (engl. Least Significant Bit*)* kuvaa vähiten merkitsevää bittiä. Kuvasta nähdään, että luvun merkitsevin bitti on etumerkkibitti *s*, seuraavat bitit kuvaavat eksponenttia ja viimeisenä esitetään luvun mantissa. [4]

## 2.2 Liukulukujen erikoisarvot ja pyöristys

Reaalilukujen joukko on ääretön, mutta liukuluvuilla voidaan esittää vain äärellinen määrä lukuja. Tästä syystä liukuluvuille on määritelty muutama erikoisarvo erittäin suurten ja pienten lukuarvojen esittämiseen. Todella pienet arvot, jotka ovat liukuluvun tarkkuuden ulkopuolella, pyöristyvät alaspäin nollaksi. Erittäin suurille arvoille taas määritellään äärettömyydet $-\infty$ ja $\infty$. Koska liukuluvuille on määritelty kaksi ääretöntä, määrittelee standardi niille myös kaksi nollaa, $-0$ ja $+0$. IEEE 754 määrittelee lisäksi vielä arvon NaN (engl. Not a Number). Tätä arvoa käytetään esimerkiksi tilanteissa, joissa laskutulosta ei ole määritelty tai sitä ei voida ilmoittaa reaalilukujen joukolla. Esimerkiksi negatiivisen luvun neliöjuuri tai kahden äärettömän yhteenlasku ovat operaatioita, joista saadaan tulokseksi NaN. [4, 5] NaN-arvoja on kahta eri tyyppiä: qNaN (engl. quiet NaN) sekä sNaN (engl. signaling NaN). Tyypit eroavat siten, että poikkeuksen tapahtuessa aritmetiikassa qNaN-arvo kulkeutuu aritmetiikan läpi, eikä ilmoita poikkeuksesta. sNaN-arvo taas ilmoittaa poikkeuksen sattumisesta, jolloin sitä voidaan käyttää hyödyksi esimerkiksi virheen paikantamisessa. Erikoisarvojen esittäminen liukuluvussa tapahtuu tietyillä eksponentin ja mantissan arvoilla. Liukuluku tulkitaan NaN-arvoksi, jos eksponentti saa suurimman arvonsa $2^w - 1$ ja mantissan arvo eroaa nollasta. Jos taas eksponentti saa suurimman arvonsa $2^w - 1$ ja mantissan arvo on nolla, tulkitaan liukuluku positiiviseksi tai negatiiviseksi äärettömäksi etumerkkibitin mukaan. Jos sekä eksponentin että

mantissan arvot ovat nollia, liukuluku on joko positiivinen tai negatiivinen nolla etumerkkibitin mukaan. [4] Taulukkoon 2 on koottu esimerkkinä 32-bittisen liukuluvun erikoisarvot, ja niitä vastaavat biasoidun eksponentin ja mantissan arvot binäärimuodossa.

Taulukko 2. *32-bittisen liukuluvun erikoisarvoja vastaavat luvut binäärimuodossa [1]*

| Erikoisarvo | Etumerkki | Eksponentti | Mantissa |
|---|---|---|---|
| $-0$ | 1 | 00000000 | 00000000000000000000000 |
| $+0$ | 0 | 00000000 | 00000000000000000000000 |
| $-\infty$ | 1 | 11111111 | 00000000000000000000000 |
| $+\infty$ | 0 | 11111111 | 00000000000000000000000 |
| NaN | 0 | 11111111 | Nollasta eroava |

IEEE 754 määrittelee liukulukujen pyöristykselle neljä eri operaatiota. Nämä ovat pyöristys nollaan, jompaankumpaan äärettömään sekä lähimpään lukuun. Liukuluvun pyöristys nollaan tapahtuu silloin, kun siinä tapahtuu *alivuoto*. Tällöin esimerkiksi aritmetiikan tuloksena on saatu luku, joka on käytettävän lukutarkkuuden alapuolella. Vastaavasti taas jos tulokseksi saadaan liukuluku, joka on käytettävän tarkkuuden ulkopuolella, tapahtuu *ylivuoto*. Tällöin luku pyöristyy jompaankumpaan äärettömään määritetyn etumerkin perusteella. Jos liukuluku on lukuskaalan sisäpuolella, mutta ei esitettävissä käytettävällä tarkkuudella, pyöristetään se lähimpään mahdolliseen lukuun. Jos pyöristettävä luku on kahden mahdollisen luvun puolessavälissä, valitaan se, jonka vähiten merkitsevä bitti on 0. [5]

# 3. LIUKULUKUYKSIKKÖ

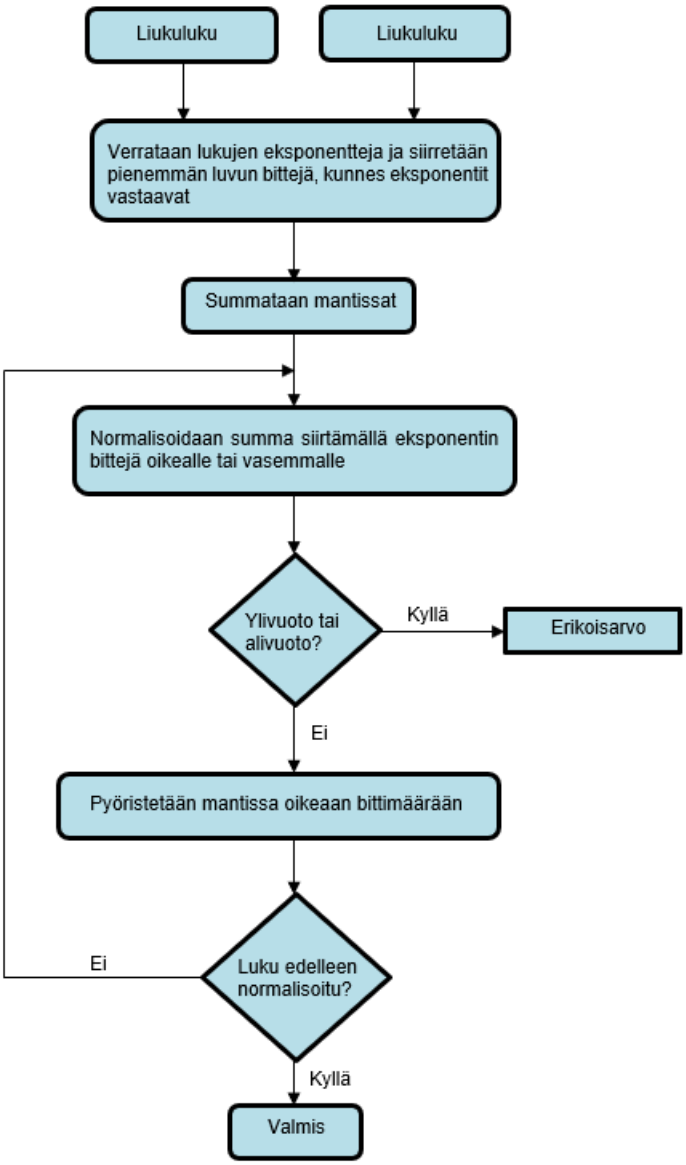
Edellä käsiteltiin liukulukuja sekä niiden tärkeyttä kaikessa tietokoneen laskennassa. Liukulukujen hyötyjä ei kuitenkaan saavuteta ilman keinoa käsitellä niitä tietokoneella. Aikanaan liukulukujen yleistyessä niiden käsittely hoidettiin ohjelmallisesti [1]. Liukulukujen monimutkainen laskenta ainoastaan ohjelmallisesti oli kuitenkin todella hidasta. Ratkaisuksi muodostuikin ainoastaan reaalilukulaskentaan tarkoitettu *liukulukuyksikkö* (engl. FPU, Floating Point Unit). Joissain lähteissä yksikköä kutsutaan myös nimillä liukulukulaskentayksikkö tai matematiikkasuoritin.

Aluksi liukulukuyksiköt toteutettiin erillisinä yksikköinä tietokoneen prosessorin rinnalle. Vielä 1980-luvullakaan samalle sirulle ei saatu riittävää määrää transistoreita, jotta sille voitaisiin toteuttaa prosessorille sekä kokonais- että reaalilukulaskenta. [6] Esimerkiksi Intelin 8087 oli ensimmäisiä liukulukuyksiköitä, jotka suunniteltiin toimivaksi erillisinä komponentteina prosessorin kanssa [7]. Erillinen liukulukuyksikkö paransi merkittävästi tietokoneen laskentatehoa, kun kaikkea laskentaa ei enää tarvinnut tehdä ohjelmallisesti. 1990-luvun alusta lähtien sirujen valmistus oli edennyt siihen pisteeseen, että nyt myös liukulukuyksikkö saatiin toteutettua prosessorin kanssa samalle sirulle. Nykyaikaisissa siruissa voi olla jopa useita liukulukuyksiköitä prosessorin rinnalla laskentatehon kasvattamiseksi. [6]

## 3.1 Liukulukujen aritmetiikka

Nykyaikaisten prosessoreiden integroidut liukulukuyksiköt tarjoavat tuen liukulukujen summaus-, erotus- ja tulo-operaatioille. Joillain liukulukuyksiköillä voidaan myös käsitellä muitakin liukulukuoperaatioita, kuten jakolaskua tai neliöjuurta. Nämä operaatiot ovat kuitenkin toteutukseltaan monimutkaisempia, jolloin ne saatetaan toteuttaa sekä ohjelmallisesti että liukulukuyksikön tukemana. [1]

Liukulukujen aritmeettiset operaatiot perustuvat useimmiten luvun eksponentin ja mantissan erilliseen käsittelyyn. Joskus nämä saattavat tapahtua myös rinnakkaisina operaatioina. Koska eksponentin ja mantissan tallettamiseen on saatavilla vain rajattu määrä bittejä, kahden liukuluvun osien käsittely vaatii yleensä niiden muokkausta esimerkiksi bittisiirtäjien avulla. [6] Kuvassa 2 on esitetty esimerkkinä vuokaaviona algoritmi liukulukujen yhteenlaskulle.




***Kuva 2.*** *Kahden liukuluvun yhteenlasku [6]*

Kuten kuvan 2 vuokaaviosta nähdään, ennen liukulukujen yhteenlaskua on eksponentteja muokattava operaation mahdollistamiseksi. Summaamisen jälkeen tulos on normalisoitava ja samalla tarkistetaan mahdollinen yli- tai alivuoto, joka johtaa tuloksen pyöristämiseen luvussa 2 esitettyihin arvoihin. Normalisoinnin jälkeen mantissa on vielä pyöristettävä oikeaan arvoon, jonka johdosta luku saatetaan joutua normalisoimaan uudelleen. [6]

## 3.2 Liukulukuyksikkö FPGA:lle

Toteutettaessa liukulukuyksikköä tarkasteltavat parametrit riippuvat kohdetarkoituksesta. Jos liukulukuyksikkö on tarkoitus toteuttaa keskusprosessorille, tärkeimmäksi parametriksi nousee operaatioiden tuottama viive. [1] Mooren lain [6] ansiosta transistoreiden vaatima pinta-ala pienenee koko ajan, jolloin keskusprosessoreissa olevan yksikön vaatima nopeus nousee käytettyä pinta-alaa tärkeämmäksi tekijäksi. Jos liukulukuyksikkö on tarkoitus toteuttaa GPU:lle (engl. Graphics Processing Unit) tai uudelleenohjelmoitavalle alustalle kuten FPGA:lle, keskitytään enemmän toteutuksen käyttämään pinta-alaan. Tällöin esimerkiksi usean rekisteritason tuottama viive ei ole FPGA-toteutuksille niin merkittävä seikka. [1]

FPGA-alustoissa yksikön toteuttamiseen käytetään uudelleenohjelmoitavia logiikkalohkoja, jotka koostuvat hakutauluista sekä muistielementeistä. Hakutaulut määrittävät lohkon antaman ulostulon sisäänmenojen perusteella totuustaulun tavoin. Logiikkalohkojen lisäksi yksikön toteuttamisessa käytetään alustojen muistielementtejä, sekä sisäänmenojen ja ulostulojen määrittämiseen käytettäviä lohkoja. [8] Nykyaikaiset FPGA-alustat sisältävät lisäksi erillisiä DSP (engl. Digital Signal Processing) -lohkoja, jotka ovat digitaalisen signaalinkäsittelyn laskentaan optimoituja lohkoja. Xilinxin 7-sarjan FPGA-alustojen DSP-lohkot sisältävät myös erillisen tulonlaskentakomponentin. [9] Xilinxin 7-sarja mukaan lukien FPGA-alustojen DSP-lohkot ovat perinteisesti suorittaneet laskentaa kiintolukujen avulla [9, 10]. Vuonna 2014 Altera esitteli ensimmäisen alustan, joka sisältää alustalle integroituja liukulukukomponentteja. Alusta sisältää summaus- sekä tulokomponentit, jotka tukevat 32-bittisiä liukulukuja. Näitä komponentteja hyväksikäyttäen Alteran alustalla voidaan suorittaa tehokkaammin suuri määrä erilaisia liukulukuoperaatioita. [10] Jos tällaiset liukulukukomponentit yleistyvät tulevaisuudessa FPGA-alustoilla, voidaan liukulukulaskentaa suorittaa niillä entistä tehokkaammin, sillä alustalle ei enää välttämättä tarvitse implementoida erillistä aritmetiikkaa.

FPGA-alustojen lohkojen toiminta ja niiden väliset yhteydet muodostetaan logiikkasynteesillä. Synteesiprosessissa laitteistonkuvauskielellä kuvattu logiikka muunnetaan matalamman tason kuvaukseksi. Seuraavaksi logiikka ohjelmoidaan alustan lohkoihin, ja lohkot vielä yhdistetään toisiinsa reitittämällä niiden väliset signaalit. Kahta viimeistä vaihetta kutsutaan implementaatioksi. [8] Suurin osa FPGA-toteutuksen viiveestä syntyykin siitä, kun alustan lohkoja reititetään toisiinsa. Nykyaikaisissa FPGA-laudoissa on kuitenkin toteutettuna niin kutsuttu fast-carry -reititys, jonka ansiosta taulut saadaan reititettyä toisiinsa nopeasti. Tällöin myös FPGA-toteutusten viivettä saadaan merkittävästi minimoitua. [1]

## 3.3 Liukulukujen aritmetiikan alkutekijät

Liukulukujen aritmeettisten operaatioiden voidaan ajatella koostuvan erilaisista alkutekijöistä, joita voidaan käyttää hyväksi operaatioita toteutettaessa. Liukulukujen operaatioiden toteutuspohjana toimivat muun muassa vastaavat aritmeettiset operaatiot kokonaisluvuille. Näihin kuuluvat summa ja erotus, kertolasku ja kokonaislukujen jakolasku jakojäännöksellä. Toteutus vaatii myös mantissan käsittelyyn tarvittavia komponentteja. Näihin kuuluu esimerkiksi bittisiirtäjiä ja laskureita. Operaatioiden toteutuksessa voidaan käyttää myös taulukoita, joissa on valmiina usein laskettuja arvoja, mikä nopeuttaa operaatioita. [1]

Eräs tärkeimmistä liukulukujen laskutoimituksiin tarvittavista komponenteista on kokonaislukujen summaaja. Kokonaislukujen summaaminen on osa kaikkia liukulukujen aritmeettisia operaatioita, mutta eri operaatiot vaativat summaamiselta eri ominaisuuksia. Esimerkiksi mantissan summaaminen vaatii komponentilta nopeutta, mikä ei ole kaikille operaatioille yhtä tärkeää. On olemassa useita kokonaislukujen summaajien arkkitehtuureita, jolloin voidaan vastata kaikkiin eri liukulukuoperaatioiden vaatimuksiin. [1] Eri summausarkkitehtuurien tarkempi tarkastelu ei kuulu tämän työn piiriin.

Liukuluvun normalisointi toteutetaan bittisiirtäjän avulla. Tällöin sen johtava ’1’-bitti tuodaan bittijonon mantissan vasempaan reunaan. Mantissan ensimmäisen ykkösbitin edellä olevia nollia kutsutaan johtaviksi nolliksi ja niiden lukumäärän laskemiseen käytetään erillistä nollalaskuria. Laskurin antama tulos annetaan parametriksi bittisiirtäjälle, joka taas luvun perusteella siirtää bittejä vasemmalle niin kauan, että ykkösbitti on mantissan vasemmassa reunassa.

Osassa liukulukuoperaatioista on suuresti hyötyä, ja joskus myös tarpeellista, jos laskuoperaatiossa hyväksikäytetään valmiita tuloksia. Varsinkin osamäärän tai neliöjuuren määrittämisessä on vaatimuksena valmiiksi laskettu osatulos, esimerkiksi arvio käänteisluvun arvosta. Näiden lisäksi monien muiden operaatioiden funktiot on määritelty tällaisissa taulukoissa, kuten esimerkiksi potenssi-, eksponentti- sekä trigonometriset funktiot. Taulukoiden käyttö FPGA-alustoilla on hyvin perusteltua, sillä alustoilla on usein saatavilla runsaasti muistia. Tällöin taulukoiden tallettaminen alustan muistipaikkoihin helpottaa merkittävästi operaatioiden toteuttamista sekä tehostaa niiden toimintaa. [1]

# 4. KOEJÄRJESTELYT

Työhön valittuja liukulukuyksiköitä vertailtiin kolmessa mittauksessa. Ensimmäisessä mittauksessa yksiköt implementoitiin valitulle FPGA-alustalle ja niiden käyttämät resurssit ja tehonkulutus kirjattiin ylös. Toisessa mittausosiossa tarkastellaan yksiköiden riippuvuutta kohdealustasta. Kolmannessa mittauksessa vertaillaan rekisteritason ja korkeamman tason toteutuksia. Tässä luvussa esitellään työhön päätyneet liukulukuyksikkötoteutukset, synteesityökalu sekä kohdealustat.

## 4.1 Valitut liukulukuyksikkötoteutukset

Liukulukuyksikkötoteutukset etsittiin avoimen lähdekoodin toteutuksina internetistä. Töiden valintaan vaikuttivat niiden toteutustapa, dokumentaatio ja verifikaatio. Työn mittauksiin valittiin kolme laitteistonkuvauskielellä toteutettua yksikköä. Lisäksi työhön valittiin yksi korkeamman tason liukulukuyksikkötoteutus, jotta voitiin vertailla eri kuvaustasoilla toteutettujen yksiköiden eroja. Seuraavaksi esitellään työhön valitut toteutukset.

### 4.1.1 Johns Hopkins Universityn Floating-Point Package

Ensimmäinen valittu liukulukuyksikkötoteutus on Johns Hopkins Universityn Github-palvelussa julkaisema lähdekoodi [11]. Yksikkö julkaistiin vuonna 2011 ja sen ovat toteuttaneet yliopiston opiskelijat henkilökunnan ohjaamana. Yksikön valintaan työhön testattavaksi vaikutti sen kattava dokumentaatio sekä sen verifikaatio. Toteutus sisältää VHDL-laitteistonkuvauskielellä toteutetun liukulukuoperaatiokirjaston, johon on toteutettu neljä liukulukujen aritmeettisiin operaatioihin käytettyä komponenttia. Komponentti *FPP_MULT* toteuttaa lukujen kertomisen*, FPP_ADD_SUB* summan ja erotuksen sekä *FPP_DIV* lukujen osamäärän. Lisäksi komponenttia *MantissaDivision* käytetään osamäärän laskemisessa mantissan käsittelyyn. Paketti sisältää myös kaksi funktiota, jotka muuttavat liukuluvun logiikkavektoriksi ja toisinpäin. Komponentit käsittelevät liukulukuja 32-bittisinä IEEE 754:n mukaisesti erikoisarvoja lukuun ottamatta. Toteutuksessa ylivuotavaa lukua ei esitetä äärettömänä, vaan se esitetään suurimpana mahdollisena lukuna. Kolme aritmetiikan sisältävää komponenttia on toteutettu tilakoneina. Komponentit suunniteltiin toimivaksi globaalilla 50 MHz:n kellolla. Toteutuksen dokumentaation mukaan se on verifioitu implementoimalla se XSA-3S1000 FPGA-laudalle, joka sisältää Xilinx XC3S1000 Spartan3 -sirun. Yksikköä ei ole testattu muilla alustoilla. Yksikössä on käytetty hyväksi Spartan3:n tulonlaskentaa tulo-operaatiota suunniteltaessa. [11] Tällöin tulo-operaatiota voidaan joutua muokkaamaan, jos yksikköä on tarkoitus

käyttää jollain toisella alustalla. Liukulukuyksikön toteutus löytyy kokonaisuudessaan tiedostosta FloatPt.vhd, joka kopioitiin sellaisenaan.

Yksikön dokumentaation mukaan sen toteutuksessa tehtiin muutamia kompromisseja. Komponentissa FPP_ADD_SUB komponentti suunniteltiin käyttämään enemmän tilaa, jotta sen toiminta saatiin tehokkaammaksi. Toteutuksessa keskityttiin myös lähinnä summaan sekä tuloon, jolloin osamäärän tehokkuuden parantamiseen ei ole kiinnitetty enempää huomiota. [11]

### 4.1.2 Lundgrenin 64-bittinen FPU

Toinen työhön valittava toteutus saatiin myös Github-palvelusta. Lähdekoodi on alun perin julkaistu opencores-palvelussa. Yksikön on toteuttanut yksityishenkilö David Lundgren ja se on julkaistu vuonna 2009. Tämänkin toteutuksen valintaan vaikuttivat sen kattava dokumentaatio sekä sen verifikaatio. Toteutettu yksikkö käsittelee liukulukuja 64-bittisinä. Liukulukuyksikkö on toteutettu VHDL-laitteistonkuvauskielellä ja se on jaettu yhdeksään eri tiedostoon. Neljässä tiedostossa on kussakin toteutettu yksi yksikön liukulukuoperaatioista. Nämä operaatiot ovat summaus tiedostossa fpu_add.vhd, erotus tiedostossa fpu_sub.vhd, tulo tiedostossa fpu_mul.vhd ja osamäärä tiedostossa fpu_div.vhd. Yksikölle on lisäksi määritelty kahdessa tiedostossa lukujen pyöristys- sekä poikkeuskäsittelyoperaatiot. Tiedosto fpu_round.vhd sisältää liululuvun pyöristysoperaatiot. Operaatiot pyöristävät luvut IEEE 754:n mukaisesti lähimpään lukuun, nollaan tai jompaankumpaan äärettömään. Tiedostoissa fpupack.vhd ja comppack.vhd on määritelty liukulukuyksikön komponenttien käyttö. Viimeinen tiedosto fpu_double.vhd toimii arkkitehtuurin ylimpänä tasona ja määrittelee yksikön sisäänmenot ja ulostulot. [12] Kaikki tiedostot kopioitiin synteesiin sellaisenaan.

Liukulukuyksikkö on suunniteltu käytettäväksi yhdellä globaalilla kellolla, jonka taajuus on 185 MHz. Toteutuksen dokumentaation mukaan yksikköä on testattu kattavasti ja sen on todettu toimivan monissa poikkeustilanteissa, kuten ali- tai ylivuodon sattuessa. Dokumentaation mukaan yksikkö on myös implementoitu Xilinxin Virtex-5 -alustalle. Dokumentaatiossa mainitaan toteutuksen olevan muita yksiköitä hitaampi, mutta tämän todetaan johtuvan kattavasta poikkeuskäsittelystä. Yksikössä on myös tietoisesti kompromissina lisätty logiikkalohkojen määrää latenssin kustannuksella. Toteutuksessa tulo-operaatio on ensimmäisen toteutuksen tavoin suunniteltu tietylle kohdealustalle, tässä tapauksessa Virtex-5:lle. Operaatiossa on hyödynnetty alustan kykyä laskea kahden mat-

riisin tuloa ja sovitettu lähdekoodi tätä ominaisuutta silmällä pitäen. [12] On siis mahdollista, että tulo-operaatiota joudutaan muokkaamaan, jos yksikköä halutaan käyttää jollain muulla alustalla.

### 4.1.3 Usselmannin 32-bittinen FPU

Kolmas valittu toteutus oli toisen toteutuksen tavoin alun perin Opencores-palvelussa julkaistu toteutus, jonka lähdekoodi oli ladattu myös Githubiin. Yksikön on toteuttanut yksityishenkilö Rudolf Usselmann. Toteutuksen suurimmat eroavaisuudet kahteen aiemmin valittuun on sen toteutuskieli ja julkaisuajankohta. Yksikkö on toteutettu Verilog-laitteistonkuvauskielellä ja se on kehitetty vuonna 2000. Yksikkö käsittelee liukulukuja 32-bittisinä IEEE 754:n mukaisesti. Yksikön arkkitehtuuri koostuu kahdesta normalisointiyksiköstä, jotka muokkaavat liukuluvun mantissaa ja eksponenttia. Nämä löytyvät tiedostoista pre_norm.v ja pre_norm_fmul.v. Yksikköön toteutetut liukulukuoperaatiot ovat summa, erotus, tulo ja osamäärä. Näiden toteutus löytyy tiedostosta primitives.v. Tiedostoon post_norm.v on toteutettu operaatioiden antaman tuloksen normalisointi. Toteutuksesta löytyy myös tiedosto exceptions.v, johon on määritelty lukujen poikkeuskäsittely liukulukustandardin mukaisesti. [13]

Toteutuksen dokumentaation mukaan sen toiminta on verifioitu simulaatioilla, mutta sen implementaatiosta FPGA:lle ei ole mainintaa. Lähdekoodista selviää toteutuksen olevan suunniteltu 10 MHz:n kellolle. [13]

### 4.1.4 FloPoCo

Edellä esitellyt liukulukuyksikkötoteutukset oli toteutettu laitteistonkuvauskielellä. Tällöin rekisteritasojen välisten signaalien käyttäytyminen ja ajoitukset on kuvattu ihmisen toimesta. Tutkimuksen kohteena on ollut myös yksiköiden toteuttaminen korkean tason synteesillä. Rekisteritason kuvauksesta poiketen korkean tason kuvaus ei ota kantaa operaatioiden ajoitukseen, vaan kuvaa ainoastaan niiden järjestyksen [14]. Tällöin ihmisen sijaan ajoitusten onnistumisen kuvaa algoritmi. Kolmannessa mittausosiossa onkin tarkoitus tarkastella implementaatiotulosten mahdollisia eroavaisuuksia näiden kahden kuvaustason välillä.

Yksi pitkään kehitetyistä korkeamman tason toteutuksista on ENS-Lyonin yliopistossa kehitetty FloPoCo-projekti. FloPoCo (engl. Floating Point Core) on C++ -ohjelmointikielellä toteutettu aritmeettisten operaatioiden generaattori. FloPoCo tarjoaa neljän perusoperaation lisäksi myös tuen esimerkiksi neliöjuurelle ja trigonometrisille funktioille, ja

sen operaatiotarjontaa laajennetaan jatkuvasti. Käyttäjä syöttää ohjelmalle halutun operaation ja ohjelma palauttaa VHDL-tiedoston, johon on kuvattu operaation toiminta. [15] Tällöin itse synteesi toteutetaan samalla tavalla kuin rekisteritason kuvauksille. FloPoCo mahdollistaa myös operaatioiden muokattavuuden ja käyttäjä voi itse määrittää komponenteissa käytettävän bittimäärän. Operaatiolle voidaan myös määrittää niiden toimintataajuus sekä kohdealusta. FloPoCo ei vielä tue Xilinxin 7-sarjaa, jolloin kohdealustana käytetään oletusalustaa Virtex-5. [15] Tämän alustan käyttö on muutenkin vertailun kannalta perusteltua, sillä myös edellä esitellyt toteutukset ovat alun perin suunniteltu vanhemmille alustoille.

Huomionarvoista FloPoCossa on sen IEEE:n standardista eroava tapa esittää liukulukuja. IEEE 754:ssä liukulukujen erikoisarvot tulkitaan eksponentin tiettyjen arvojen avulla, kun taas FloPoCon käyttämässä formaatissa erikoisarvot esitetään erillisten bittien avulla liukulukubittijonon alussa [4]. Tämän esitystavan etuna on säästö erikoisarvojen tulkintaan kuluvassa logiikassa. Esitystavan haittapuolena FloPoCo-formaatin liukuluku vaatii kaksi bittiä enemmän kuin IEEE:n standardissa. FloPoCo ei myöskään tue ei-normalisoituja lukuja. FloPoCon käyttämän esitystavan eroaminen standardista ei ole ongelma, sillä se tarjoaa myös operaatiot liukuluvun muuntamiseen esitystavasta toiseen. [15]

Vertailua varten työhön generoidaan FloPoCo:lla samat aritmeettiset operaatiot kuin edellä esitetyissä laitteistonkuvauskielitoteutuksissa. Nämä operaatiot ovat summa, erotus, tulo ja osamäärä. Operaatiot generoitiin toimimaan 64-bittisinä ja 185 MHz:n taajuudella. Tällöin generoituja operaatioita voidaan vertailla Lundgrenin 64-bittisen yksikön kanssa. Aritmeettisten operaatioiden lisäksi generoidaan operaatiot, jotka muuttavat liukuluvut IEEE 754:n mukaisesta esitystavasta FloPoCon esitystapaan ja toisinpäin.

## 4.2 Vivado Design Suite

Valittujen liukulukuyksiköiden implementaatio FPGA-alustalle toteutettiin Xilinxin Vivado Design Suite -ohjelmistolla. Vivado on logiikkasynteesityökalu, jonka avulla voidaan syntetisoida, implementoida ja analysoida laitteistonkuvauskielitoteutuksien toimintaa. Ohjelmisto tarjoaa myös edistyneitä työkaluja, kuten korkean tason synteesiä tai SoC (engl. System on Chip) -suunnittelua varten. Vivado toteuttaa toteutuksissa kuvatun logiikan FPGA-alustalle ja reitittää signaalit kulkemaan optimaalisesti. Vivadon avulla voidaan tarkastella toteutusten käyttämiä resursseja, tehonkulutusta sekä ajoituksia. [16] Synteesit toteutettiin luomalla jokaiselle toteutukselle Vivadoon oma projektinsa, johon lisättiin

toteutuksen lähdekoodi. Projektiin lisättiin myös kohdealusta, jolle toteutus implementoitiin. Valitut FPGA-alustat on esitelty kappaleessa 4.3.

Valittujen liukulukuyksikkötoteutusten dokumentaatioissa on esitelty kellotaajuudet, joilla ne on suunniteltu toimiviksi. Vivadon synteesiasetuksista on mahdollista generoida virtuaalinen kellosignaali toteutusten käytettäväksi [17]. Tällainen kello generoitiin jokaiseen projektiin, jolloin implementaatioista saatiin tulokseksi myös ajoitustietoja. Vivado mittaa synteeseistä *slackia*, joka kertoo eron vaaditun ja toteutuneen ajoituksen välillä [18]. Vivado tarjoaa implementaatiotuloksina WNS:n (engl. Worst Negative Slack) ja WHS:n (engl. Worst Hold Slack), jotka kertovat huonoimman slackin toteutuksen ajoitukselle [19]. Kun rekisterikiikku lukee uutta dataa, sen on oltava muuttumattomana tietyn ajan ennen kellon nousevaa reunaa (set-up time) [20]. WNS kuvaa datan todellisen valmiinaoloajan ja vaaditun ajan erotusta. Jos arvo on negatiivinen, toteutuksen ajoitus epäonnistuu, sillä rekisterikiikut lukevat keskeneräistä dataa. Arvon ollessa positiivinen se kertoo ajan, jonka data on valmiina luettavaksi ennen kuin kiikku yrittää sitä lukea. [19] Kiikun sisäänmenosignaalin on pysyttävä muuttumattomana myös tietty aika kellosignaalin nousevan reunan jälkeen (hold time) [20]. WHS kuvaa erotusta todellisen arvon pitoajan sekä vaaditun ajan välillä. WNS:n tavoin toteutuksen ajoitus epäonnistuu, jos WHS:n arvo on negatiivinen. Tällöin rekisterikiikku ei saa annettua oikeaa arvoa eteenpäin. [19]

Vivado tarjoaa FPGA-alustalle implementoidulle toteutukselle myös tehonkulutusarvion. Arvio on kaikkein tarkin sen jälkeen, kun työkalu on suunnitellut logiikan reitityksen sekä resurssienkäytön alustalle. On huomioitavaa, että Vivadon tarjoama tieto perustuu vain laskennalliseen arvioon ja todellinen tehonkulutus selviää vasta fyysiseltä alustalta tehtävistä mittauksista. [21]

Vivadossa projektin lähdekoodi on järjestetty hierarkkisesti. Toteutuksen ylin taso voidaan määrittää manuaalisesti, jolloin ohjelma syntetisoi toteutuksesta ainoastaan osat, jotka ovat hierarkkisesti sen alapuolella. Asettamalla toteutuksesta haluttu liukulukuoperaatio ylimmäksi tasoksi saadaan synteesin tuloksena ainoastaan kyseisen operaation tiedot. [22] Tällöin vaihtelemalla toteutuksen hierarkiatasoa saatiin implementoitua jokainen komponentti erikseen. Lundgrenin ja Usselmannin toteutuksissa on myös toteutettu erillinen ylätaso liukulukuyksikölle, jolloin tällä tasolla syntetisoimalla saatiin implementaatiotulokset koko yksikölle.

## 4.3 Valitut FPGA-alustat

Työssä päätettiin käyttää Xilinxin valmistamia FPGA-alustoja, sillä ne ovat hyvin tuettuina Vivadossa. Ensimmäisen osan mittauksiin valittiin Xilinxin Virtex-7 -alusta vuodelta 2010. Valmistajan Virtex-sarjaan kuuluvat sen tehokkaimmat ja suorituskykyisimmät mallit, joissa on saatavilla suuri määrä resursseja ja laskentatehoa. [23] Valitun Virtex-7 -alustan osanumero Vivadossa on xc7vx485tffg1761-1.

Toisessa mittausosiossa toteutuksia mitataan eri alustoilla. Yksi valituista alustoista on Xilinxin Spartan-7 -alusta vuodelta 2010. Valmistajan Spartan-sarja edustaa alustojen halvempaa hintaluokkaa, jossa on tähdätty matalaan tehonkulutukseen ja pieneen kokoon. [24] Valitun Spartan-7 -alustan osanumero Vivadossa on xc7s100fgga676-1. Mittaukseen valittiin myös kolmanneksi alustaksi Xilinxin Virtex UltraScale+. Alkuvuodesta 2016 julkaistu FPGA-alusta on valmistajan uusin ja tehokkain ja se tarjoaa suuren määrän logiikkalohkoja sekä muistia. [25] Valitun Virtex UltraScale+ -alustan osanumero Vivadossa on xcvu9p-flga2104-2L-e.

# 5. TULOKSET

## 5.1 Laitteistonkuvauskielitoteutusten implementaatiotulokset

Johns Hopkins Universityn toteutuksen mukaiset liukuluvunkäsittelykomponentit implementoitiin valitulle FPGA-alustalle käyttäen 50 MHz:n virtuaalista kelloa. Taulukkoon 3 on koottu toteutuksen aritmeettisten operaatioiden käyttämät resurssit ja tehonkulutus.

Taulukko 3. *Johns Hopkins Universityn liukulukuyksikönoperaatioiden implementaatiotulokset*

| Resurssi | FPP_ADD_SUB | FPP_MULT | FPP_DIV |
|---|---|---|---|
| Hakutaulut (kpl) | 702 | 41 | 244 |
| Kiikut (kpl) | 130 | 119 | 271 |
| Puskurit (kpl) | 1 | 1 | 2 |
| Tehonkulutus (W) | 0,010 | 0,006 | 0,003 |

Operaatioiden resurssienkulutuksia tarkastelemalla huomataan summaus- ja erotusoperaation käyttävän suurimman määrän hakutauluja, mutta toisaalta vaativan vähemmän kiikkuja, eli rekisteritasoja. Osamäärän toteuttava operaatio vaatii yhden rekisteripuskurin enemmän, jotta rekisterien välinen viive kaikilla reiteillä pysyisi samansuuruisena. Operaatioiden tehonkulutuksen todettiin olevan samaa suuruusluokkaa. Johns Hopkins Universityn toteutuksessa aritmeettiset operaatiot on toteutettu erillisinä komponentteina eikä niitä ole koottu yhteen liukulukuyksikköön. Tällöin ei voida implementoida koko yksikön resurssien- ja tehonkulutusta.

Lundgrenin 64-bittinen yksikkö implementoitiin kohdealustalle käyttäen 185 MHz virtuaalista kellotaajuutta. Ensimmäisestä toteutuksesta poiketen implementaation tuloksena saatiin myös koko liukulukuyksikön käyttämät resurssit sekä tehonkulutus. Toteutuksen liukulukuyksikön sekä operaatioiden implementaatiotulokset on koottu taulukoihin 4 ja 5.

Taulukko 4. *Lundgrenin liukulukuyksikön ja sen operaatioiden implementaatiotulokset*

| Resurssi | yksikkö | summa | erotus | tulo |
|---|---|---|---|---|
| Hakutaulut (kpl) | 3478 | 368 | 632 | 1063 |
| Kiikut (kpl) | 4233 | 672 | 678 | 944 |
| Puskurit (kpl) | 1 | 1 | 1 | 1 |
| Tehonkulutus (W) | 0,116 | 0,119 | 0,127 | 0,190 |

Taulukko 5. *Lundgrenin liukulukuyksikön operaatioiden implementaatiotulokset*

| Resurssi | osamäärä | pyöristys | poikkeuskäsittely |
|---|---|---|---|
| Hakutaulut (kpl) | 920 | 61 | 241 |
| Kiikut (kpl) | 912 | 258 | 419 |
| Puskurit (kpl) | 1 | 1 | 1 |
| Tehonkulutus (W) | 0,067 | 0,065 | 0,047 |

Taulukossa 3 esitettyjä implementaatiotuloksia tarkasteltaessa huomataan tulo-operaation olevan resurssinkulutukseltaan kaikkein vaativin operaatio. Se vaatii kaikkein eniten sekä hakutauluja että kiikkuja. Tulo on myös yksittäisistä operaatioista suurin tehonkulutukseltaan. Pelkän tulo-operaation tehonkulutus on jopa suurempi kuin koko liukulukuyksikön. Tämä saattaa johtua Vivadon kyvystä optimoida logiikan reititystä FPGA-alustalle kokonaisen yksikön implementaation kohdalla. Vivadon antama tehonkulutus on myös vain arvio ja todellinen tehonkulutus fyysisellä FPGA-alustalla saattaa erota mittaustuloksista. Kun verrataan Lundgrenin yksikön resurssien- ja tehonkulutusta Johns Hopkins Universityn toteutukseen, huomataan Lundgrenin yksikön kuluttavan jokaisen operaation kohdalla enemmän sekä resursseja että tehoa. Tämä on johdonmukaista, sillä se käsittelee liukulukuja suuremmalla tarkkuudella sekä toimii huomattavasti nopeammalla kellotaajuudella.

Usselmannin 32-bittinen FPU implementoitiin kohdealustalle 10 MHz:n virtuaalisella kellotaajuudella. Taulukoihin 6 ja 7 on koottu toteutuksen implementaatiotulokset.

Taulukko 6. *Usselmannin yksikön ja sen aritmeettisten operaatioiden implementaatiotulokset*

| Resurssi | yksikkö | summa/ erotus | tulo | osamäärä |
|---|---|---|---|---|
| Hakutaulut (kpl) | 4178 | 28 | - | 2549 |
| Kiikut (kpl) | 532 | - | 17 | 148 |
| Puskurit (kpl) | 2 | - | 1 | 1 |
| DSP-lohkot (kpl) | 2 | - | 2 | - |
| Tehonkulutus (W) | 0,010 | 0,002 | 0,005 | 0,014 |

Taulukko 7. *Usselmannin yksikön normalisointioperaatioiden implementaatiotulokset*

| Resurssi | esi-normali-sointi | tulon esi-normalisointi | jälkinor-malisointi | poikkeus-käsittely |
|---|---|---|---|---|
| Hakutaulut (kpl) | 270 | 74 | 956 | 29 |
| Kiikut (kpl) | 71 | 16 | - | 22 |
| Puskurit (kpl) | 1 | 1 | - | 1 |
| DSP-lohkot (kpl) | - | - | - | - |
| Tehonkulutus (W) | 0,004 | 0,003 | 0,003 | 0,001 |

Usselmannin toteutuksen huomataan eroavan kahdesta aiemmasta, sillä se käyttää hyväksi myös Virtex-7:n DSP-lohkoja. Näitä lohkoja käytetään yksikön tulo-operaatiossa. Kuten luvussa 3.2 esiteltiin, nämä DSP-lohkot sisältävät erillisen tulonlaskentakomponentin. Vivadon alustalle implementoima toteutus käyttää hyväksi näitä komponentteja, jolloin se ei tarvitse käyttöönsä lainkaan hakutauluja. Ei ole selvää, miksi ainoastaan Usselmannin toteutus optimoidaan käyttämään näitä lohkoja. Tämä saattaa kenties johtua sen käyttämästä kellotaajuudesta, joka on huomattavasti muita toteutuksia matalampi. Operaatioista osamäärä käyttää merkittävästi enemmän resursseja kuin muut operaatiot. Toteutuksen summaus- ja erotuslohko on toteutettu täysin kombinatorisesti eikä siihen ole toteutettu rekistereitä. Tällöin operaatio käyttää resursseista ainoastaan hakutauluja. Koska operaatio ei toiminut kellosignaalin mukaan, Vivado ei myöskään tuottanut operaatiolle tehonkulutusarviota. Ongelma ratkaistiin lisäämällä synteesiin generoidun kellon varassa toimiva sisäänmenoviive [15]. Viiveen avulla kombinatorisen lohkon sisäänmenosignaalit saadaan toimimaan kellon tahdissa, jolloin implementaatiosta saadaan operaatiolle myös tehonkulutusarvio. Aritmetiikan jälkeinen jälkinormalisointilohko on toteutettu myös ilman rekistereitä, jolloin sen toteutus ei käytä kiikkuja.

Ensimmäisessä toteutuksessa ei ollut mahdollista implementoida koko liukulukuyksikön käyttämiä resursseja, vaan oli tyydyttävä yksittäisten operaatioiden tuloksiin. Kun tarkastellaan muiden toteutusten käyttämiä resursseja, huomataan yksittäisten operaatioiden resurssien summan olevan samaa luokkaa koko yksikön käyttämien resurssien kanssa. Luku ei ole täsmälleen sama, sillä operaatioiden välinen kommunikaatio vaatii myös resursseja. Tällöin jos ensimmäisen toteutuksen liukulukuoperaatiot koottaisiin yhdeksi yksiköksi, voidaan yksikön resurssienkäytölle laskea karkea arvio summaamalla yksittäisten operaatioiden resurssit yhteen.

## 5.2 Laitteistonkuvauskielitoteutusten implementaatio eri alustoille

Toisessa mittausosiossa liukulukuyksiköitä implementoitiin eri FPGA-alustoille ja tarkasteltiin niiden ajoituksia. Valitut kohdealustat on kuvattu kappaleessa 4.3. Johns Hopkins Universityn toteutuksessa liukulukuoperaatioita ei ole koottu yhdeksi yksiköksi, jolloin toteutuksesta päätettiin implementoida komponentti FPU_ADD_SUB, sillä se käyttää eniten resursseja rekisterien välillä. Muista toteutuksista implementoitiin koko yksikkö. Taulukkoon 8 on koottu Johns Hopkins Universityn toteutuksen slack-arvot eri alustoilla.

Taulukko 8. *Johns Hopkins Universityn yksikön slack-arvot eri alustoilla*

| Slack | Virtex-7 | Spartan-7 | Virtex UltraScale+ |
|---|---|---|---|
| WNS (ns) | 12,385 | 10,726 | 14,073 |
| WHS (ns) | 0,164 | 0,205 | 0,046 |

Tarkasteltaessa toteutuksen ajoituksia huomataan niissä eroa riippuen siitä, mille alustalle toteutus on implementoitu. Kun alusta vaihdetaan tehokkaampaan, signaalien kulkema matka hakutaulujen välillä on lyhyempi, jolloin toteutuksen WNS kasvaa. Tämä johtuu siitä, että data on nopeammin valmiina kiikkujen luettavaksi ennen kellosignaalin nousevaa reunaa. Toteutuksen WHS-arvoissa huomataan tapahtuvan vastakkainen ilmiö. Tämä johtunee todennäköisesti siitä, että rekistereiden välimatkan pienentyessä kiikkujen sisäänmenoarvo vaihtuu nopeammin ja se pysyy kellon nousevan reunan jälkeen lyhyemmän aikaa muuttumattomana.

Lundgrenin 64-bittisen yksikön ajoitustiedot eri alustoilla on koottu taulukkoon 9.

Taulukko 9. *Lundgrenin toteutuksen slack-arvot eri alustoilla*

| Slack | Virtex-7 | Spartan-7 | Virtex UltraScale+ |
|---|---|---|---|
| WNS (ns) | 0,569 | 0,235 | 1,207 |
| WHS (ns) | 0,080 | 0,033 | 0,013 |

Lundgrenin toteutuksen ajoituksissa huomataan samanlaisia tuloksia kuin ensimmäisen kohdalla. Kun kohdealusta vaihdetaan edistyneempään, rekistereiden välinen WNS kasvaa. Ensimmäisestä toteutuksesta poiketen Virtex-7 -alusta tarjoaa paremman WHS-arvon kuin Spartan-7.

Usselmannin 32-bittisen toteutuksen ajoitukset on koottu taulukkoon 10.

Taulukko 10. *Usselmannin toteutuksen slack-arvot eri alustoilla*

| Slack | Virtex-7 | Spartan-7 | Virtex UltraScale+ |
|---|---|---|---|
| WNS (ns) | 13,541 | -25,974 | 53,239 |
| WHS (ns) | 0,131 | 0,073 | 0,011 |

Usselmannin yksikön ajoitustulokset mukailevat kahta aiempaa ja jälleen huomataan WNS-arvojen kasvavan alustan tehokkuuden kasvaessa. Tuloksissa huomioitavaa on myös se, että vaatimattomimmalla Spartan-7 -alustalla WNS-arvo on negatiivinen. Kuten luvussa 4.2 todettiin, tällöin toteutuksen ajoitus epäonnistuu, eikä signaali ehdi kulkea rekisteristä toiseen yhden kellojakson aikana. Toteutuksen WHS-arvot eri alustoilla mukailevat toisen toteutuksen vastaavia arvoja.

Toteutusten käyttämien resurssien käytössä ei havaittu merkittäviä muutoksia alustan vaihtuessa. Alustan vaihtuessa hakutaulujen käyttö lisääntyi suurimmillaan 80:llä, joka ei pienimmälläkään alustalla tarkoita kuin noin promillen muutosta. Muiden resurssien käytössä ei havaittu muutoksia.

Taulukkoon 11 on koottu toteutusten tehonkulutukset alustoittain. Taulukossa on myös ilmoitettu toteutuksen osuus koko alustan tehonkulutuksesta.

Taulukko 11. *Toteutusten tehonkulutukset eri alustoilla*

| | | Virtex-7 | Spartan-7 | Virtex UltraScale+ |
|---|---|---|---|---|
| Johns Hopkins University | Tehonkulutus (W) | 0,010 | 0,009 | 0,018 |
| | Osuus kokonais-kulutuksesta (%) | 4 | 9 | 1 |
| Lundgren | Tehonkulutus (W) | 0,116 | 0,118 | 0,160 |
| | Osuus kokonais-kulutuksesta (%) | 32 | 55 | 6 |
| Usselmann | Tehonkulutus (W) | 0,010 | 0,010 | 0,020 |
| | Osuus kokonais-kulutuksesta (%) | 4 | 9 | 1 |

Tarkasteltaessa taulukon 11 tehonkulutuksia huomataan tehonkulutuksen kasvavan siirryttäessä tehokkaammalle kohdealustalle. Huomionarvoista on myös se, että tehokkaammilla alustoilla itse toteutuksen tehonkulutuksen merkittävyys pienenee. Esimerkiksi Virtex UltraScale+ -alustalla Johns Hopkins Universityn toteutuksen tehonkulutus kattaa enää yhden prosentin koko alustan tehonkulutuksesta. Tällöin voidaan todeta,

että on merkittävämpää toteutuksen tehonkulutuksen sijaan tarkastella kohdealustan tehonkulutusta.

## 5.3 Korkeamman tason toteutuksen synteesi

Kolmannessa mittausosiossa generoitiin FloPoColla neljä liukulukujen aritmeettista operaatiota. Operaatiot implementoitiin Virtex-7 -alustalle. Taulukkoon 12 on koottu generoitujen operaatioiden implementaatiotulokset.

Taulukko 12. *FloPoCon aritmeettisten operaatioiden implementaatiotulokset*

| Resurssi | summa | summa/erotus | tulo | osamäärä |
|---|---|---|---|---|
| Hakutaulut (kpl) | 649 | 968 | 413 | 3290 |
| Kiikut (kpl) | 517 | 823 | 365 | 4767 |
| Puskurit (kpl) | 1 | 1 | 1 | 1 |
| DSP (kpl) | - | - | 12 | - |
| Jaettu muisti (kpl) | 4 | 12 | - | 91 |
| Tehonkulutus (W) | 0,157 | 0,291 | 0,173 | 0,409 |

Taulukossa 12 esitettyjä resurssienkulutuksia tarkasteltaessa huomataan osamäärän olevan resurssitarpeiltaan vaativin. Se käyttää huomattavasti enemmän sekä hakutauluja että kiikkuja implementaatiossaan. Operaatio käyttää myös jonkin verran jaettua muistia, mikä merkitsee hakutaulujen käyttämistä muistielementtinä [26]. Osamäärä on myös selkeästi tehonkulutukseltaan suurin. Osamäärän lisäksi myös summa- ja erotusoperaatiot käyttävät hakutauluja muistina. Tulo on operaatioista ainoa, joka käyttää hyväkseen Virtexin DSP-lohkoja implementaatiossaan. Tämän seurauksena se käyttää hakutauluja muita operaatiota vähemmän. Summausoperaatio voidaan generoida myös samaan komponenttiin yhdessä erotuksen kanssa. Tällöin hakutaulujen käyttö kasvaa noin 49 %, sekä kiikkujen käyttö noin 59 %. Lisäksi jaetun muistin käyttö kolminkertaistuu ja tehonkulutus lähes kaksinkertaistuu. Taulukossa 12 esitettyjen operaatioiden lisäksi syntetisoitiin liukulukujen esitystavan muuttamiseen tarvittavat operaatiot, jotka kuluttivat ainoastaan hakutauluja. IEEE:n standardin mukaisen liukuluvun muuntaminen FloPoCon esitystapaan kuluttaa 45 hakutaulua, ja päinvastainen operaatio vaatii 64 hakutaulua.

Edellä on analysoitu FloPoCon operaatioiden resurssienkulutusta ja vertailtu niitä keskenään. Mielenkiintoisempaa on kuitenkin verrata taulukon 12 tietoja rekisteritason toteutukseen. Taulukoissa 4 ja 5 esiteltiin Lundgrenin 64-bittisen rekisteritasokuvauksen implementaatiotulokset, jotka toimivat samalla taajuudella kuin FloPoColla generoidut

komponentit. Kun toteutuksien summausoperaatioiden implementaatiotuloksia verrataan keskenään, huomataan FloPoCon generoiman operaation käyttävän huomattavasti enemmän hakutauluja. FloPoCo käyttää hieman vähemmän kiikkuja, mutta tämä johtunee todennäköisesti siitä, että se käyttää myös hakutauluja muistina. Tarkasteltaessa toteutusten tulo-operaatioiden resurssienkulutusta, huomataan FloPoCon operaation käyttävän huomattavasti vähemmän sekä hakutauluja että kiikkuja. Toisaalta FloPoCon tulo-operaatio käyttää hyväkseen Virtex-7:n DPS-lohkoja, joka todennäköisesti vaikuttaa merkittävästi muiden resurssien käyttöön. Vertailtaessa toteutusten osamääräoperaatioita huomataan FloPoCon resurssienkulutuksen olevan moninkertainen rekisteritasokuvauksen operaatioon verrattuna. Jokaisen operaation kohdalla FloPoCon tehonkulutus on myös merkittävästi suurempi.

FloPoCon ja Lundgrenin 64-bittisen toteutuksen vertailun motivaationa oli tarkastella, voidaanko korkeamman tason toteutuksella saavuttaa resurssien- ja tehonkulutukseltaan yhtä tehokas liukulukuyksikkö kuin rekisteritason kuvauksella. Rekisteritasokuvauksen suunnitteluprosessi on korkeamman tason toteutukseen verrattuna työläämpi, sillä se vaatii yksikön loogisen toiminnan lisäksi ajoituksen tarkkaa suunnittelua. Tällöin jos korkeamman tason kuvauksella voidaan toteuttaa vertailukelpoinen liukulukuyksikkö, voidaan rekisteritasokuvauksen merkittävyys kyseenalaistaa. Vertailun perusteella vaikuttaa siltä, että FloPoCo ei kykene vielä kilpailemaan hyvin suunnitellun rekisteritason kuvauksen kanssa perusoperaatioiden toteutuksessa. Jokaisen vertaillun operaation kohdalla tuloa lukuun ottamatta FloPoCon generoimat komponentit vaativat enemmän resursseja implementaatioonsa FPGA-alustalle. FloPoCon komponentit olivat myös tehonkulutukseltaan suurempia. Tässä työssä tehdyt vertailut ottivat kuitenkin kantaa ainoastaan neljän eri perusoperaation toteutukseen. FloPoCo tarjoaa näiden operaatioiden lisäksi muitakin liukulukujen käsittelyyn tarvittavia operaatioita, jotka ovat toteutukseltaan monimutkaisempia. FloPoCo tarjoaa myös generoitavalle komponentille laajan muokattavuuden, kuten esimerkiksi toimintataajuuden määrittämisen. Tällöin tämän työn mittausten perusteella ei voida todeta rekisteritason toteutuksen olevan selkeästi parempi vaihtoehto. Tulevaisuuden tutkimusaihe voisikin olla FloPoCon sekä muiden korkean tason kuvausten laajempi tutkiminen, sillä ne tarjoavat monimutkaisempia aritmeettisia operaatioita, joita löydetyt avoimen lähdekoodin rekisteritasokuvaukset eivät tue.

# 6. YHTEENVETO JA PÄÄTELMÄT

Tässä työssä etsittiin avoimen lähdekoodin liukulukuyksikkötoteutuksia. Työhön valittiin kolme laitteistonkuvauskielitoteutusta, jotka eroavat käsittelytarkkuudeltaan ja toimintataajuudeltaan. Uusimman ja vanhimman toteutuksen välillä on myös kulunut merkittävän paljon aikaa. Työhön valittiin myös yksi edelleen kehitteillä oleva korkeamman kuvaustason yksikkötoteutus. Yksiköitä vertailtiin kolmessa eri mittausosiossa. Implementaatioita varten työhön valittiin kolme resurssitarjonnaltaan eroavaa FPGA-alustaa.

Ensimmäisessä mittausosiossa yksiköitä implementoitiin samalle alustalle. Ensimmäisen mittausosion tuloksia vertailtaessa huomataan Lundgrenin liukulukuyksikkötoteutuksen olevan tehonkulutukseltaan suurin ja vaativan myös suuren määrän resursseja. Tämä todennäköisesti johtuu sen suuremmasta tarkkuudesta verrattuna kahteen muuhun toteutukseen. Lundgrenin toteutuksessa liukuluku vaatii muihin verrattuna kaksinkertaisen määrän bittejä luvun esittämiseen sekä toimii huomattavasti suuremmalla kellotaajuudella, jolloin on johdonmukaista sen käyttävän enemmän tehoa ja resursseja. Poikkeuksena Usselmannin toteutus käyttää enemmän hakutauluja kuin Lundgrenin toteutus. Tämä saattaa johtua toteutuksen aritmetiikan suunnittelusta. Kun verrataan Lundgrenin ja Usselmannin toteutuksia, huomataan jälkimmäisen käyttävän osamäärä-operaatioon huomattavan määrän resursseja. Tällöin herää kysymys johtuuko ero ainoastaan kyseisen operaation toteutuksen heikkoudesta. On myös huomioitavaa, että toteutukset on kirjoitettu eri kuvauskielillä ja niiden välillä on kulunut paljon aikaa. Laitteistonkuvauskielet ovat kehittyneet tällä aikavälillä, jolloin vanhempaa toteutusta voisi ehkä olla mahdollista optimoida nykyaikaisilla menetelmillä. Tätä poikkeusta lukuun ottamatta ensimmäisen mittausosion tuloksista voidaan päätellä yksiköiden tehon- ja resurssienkulutuksen olevan verrannollinen niiden tarkkuuteen ja kellotaajuuteen.

Toisessa mittausosiossa vaihdeltiin implementaatioiden kohdealustaa. Toisen mittausosion tulosten mukaan toteutusten ajoitukset paranivat kohdealustan vaihtuessa tehokkaampaan. Samoin huomattiin ajoitusten voivan myös epäonnistua alustan vaihtuessa heikompaan. Kohdealustan vaihtuessa tehokkaampaan huomattiin myös tehonkulutuksen kasvavan. Toteutusten tehonkulutuksen muutosta merkittävämmäksi havainnoksi nousi kuitenkin toteutuksen osuus kokonaiskulutuksesta. Tehokkaammalla alustalla itse alusta kuluttaa toteutuksia huomattavasti enemmän tehoa ainoastaan toimiakseen. Tällöin voidaan kyseenalaistaa, onko pienen toteutuksen syntetisointi suurelle alustalle

enää järkevää. Jos toteutus saadaan ajoituksen kannalta toimimaan halutulla kellotaajuudella pienemmällä alustalla, tehonkulutuksen suurentaminen ei ole välttämättä enää perusteltua. Kohdealustan valinta onkin syytä tehdä jokaisen toteutuksen kohdalla erikseen.

Kolmannessa mittausosiossa verrattiin rekisteritason ja korkeamman tason kuvauksien implementaatiotuloksia samalla alustalla. Tämän mittauksen motivaationa oli tarkastella, voidaanko korkean tason kuvauksen avulla saavuttaa yhtä tehokas liukulukuyksikkötoteutus kuin tarkempaa suunnittelua vaativalla rekisteritason kuvauksella. Mittaustulosten mukaan korkean tason synteesi vaatii enemmän resursseja operaatioiden toteuttamiseen, sekä kuluttaa myös enemmän tehoa. On kuitenkin huomioitava, että FloPoCon tarjoamista operaatioista mitattiin ainoastaan murto-osaa. Laajempi tutkimus on siis tarpeen, jotta voidaan paremmin tarkastella korkeamman tason mahdollisia etuja, jotka saattavat esiintyä vasta monimutkaisempia operaatioita toteutettaessa.

Kuten jo aiemmin todettiin, liukulukujen merkitys ei ole poistumassa. FPGA-alustojen sekä niiden työkalujen kehittyessä alustat tarjoavat jatkossakin tehokkaan ja joustavan pohjan tutkimus- ja kehitystyölle. FPGA-toteutukset liukulukujen aritmetiikalle ovat siis todennäköisesti mielenkiinnonkohde monille tulevaisuudessakin.

# LÄHTEET


[1] J.M. Muller, N. Brisebarre, F. de Dinechin, C-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, S. Torres, Handbook of Floating Point Arithmetic, Birkhäuser Boston, 2010

[2] 754-1985 -IEEE Standard for Binary Floating-Point Arithmetic, IEEE, 1985. Saatavissa: https://ieeexplore.ieee.org/document/30711. Viitattu: 14.4.2020

[3] 754-1987 -IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE, 1987. Saatavissa: https://standards.ieee.org/standard/854-1987. Viitattu: 14.4.2020

[4] 754-2008 -IEEE Standard for Radix-Independent Floating-Point Arithmetic, IEEE, 2008. Saatavissa: https://ieeexplore.ieee.org/document/4610935. Viitattu: 14.4.2020

[5] R.P. Brent, P. Zimmermann, Modern Computer Arithmetic, Cambridge University Press, 2011

[6] D.A. Patterson, J.L. Hennessy, Computer Organization and Design the Hardware/Software Interface, 4th ed., Morgan Kaufmann Publishers, c2009

[7] Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, Intel, 1999. Saatavissa: https://www.cs.cmu.edu/~410/doc/intel-arch.pdf. Viitattu: 14.4.2020

[8] J. Deschamps, G. Bioul, G. Sutter, Synthesis of arithmetic circuits FPGA, ASIC and embedded systems, Hoboken, N.J., 2005

[9] 7 Series DSP48E1 Slice, User Guide, Xilinx Inc., 2018. Saatavissa: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf. Viitattu: 15.5.2020

[10] The Industry's First Floating-Point FPGA, Altera Corporation, 2014. Saatavissa: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/bg-floating-point-fpga.pdf. Viitattu: 14.4.2020

[11] R. Fay, A. Hsieh, D. Jeang, B. Jenkins, Synthesizable VHDL Floating-Point Package, John Hopkins University ECE department, 2011. Saatavissa: https://github.com/xesscorp/Floating_Point_Library-JHU. Viitattu: 14.4.2020

[12] D. Lundgren, FPU Double VHDL, 2009. Saatavissa: https://github.com/freecores/fpu_double. Viitattu: 14.4.2020

[13] R. Usselmann, Open Floating Point Unit, 2000. Saatavissa https://github.com/freecores/fpu. Viitattu: 14.4.2020

[14] J.M.P. Cardoso, M. Weinhardt, High-level synthesis. In: D. Koch, F. Hannig, D. Ziener (eds) FPGAs for Software Programmers. Springer, Cham, 2016

[15] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. IEEE Design & Test of Computers, 28(4):18--27, 2011. Saatavissa: http://perso.citi-lab.fr/fdedinec/recherche/publis/2011-DaT-FloPoCo.pdf. Viitattu: 10.5.2020

[16] Vivado Design Suite Tutorial Using Constraints, Xilinx Inc., v2012.2, 2012. Saatavissa: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug945-vivado-using-constraints-tutorial.pdf. Viitattu: 14.4.2020

[17] Vivado Design Suite Overview. Saatavissa: https://www.xilinx.com/products/design-tools/vivado.html. Viitattu: 14.4.2020.

[18] S. Hauck, A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation, 2010

[19] Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906), Xilinx Inc., v2015.2, 2015. Saatavissa: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug906-vivado-design-analysis.pdf. Viitattu: 14.4.2020

[20] J. Crisp, Introduction to digital systems, Boston, 2000

[21] Vivado Design Suite User Guide: Power Analysis and Optimization, Xilinx Inc., v2018.3, 2018. Saatavissa: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug907-vivado-power-analysis-optimization.pdf. Viitattu: 14.4.2020

[22] Vivado Design Suite User Guide: Synthesis, Xilinx Inc., v2019.2, 2019. Saatavissa: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf. Viitattu: 14.4.2020

[23] Virtex-7 FPGA FamilyProduct Brief, Xilinx Inc., 2012. Saatavissa: https://www.xilinx.com/support/documentation/product-briefs/virtex7-product-brief.pdf. Viitattu: 14.4.2020

[24] 7 Series FPGAs Data Sheet: Overview, Xilinx Inc., 2018. Saatavissa: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. Viitattu: 14.4.2020

[25] UltraScale Architecture and Product Data Sheet: Overview, Xilinx Inc., 2019. Saatavissa: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Viitattu: 14.4.2020

[26] 7 Series FPGAs Configurable Logic Block, User Guide, Xilinx Inc., 2016. Saatavissa: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf. Viitattu: 15.5.2020