TAMPEREEN TEKNILLINEN YLIOPISTO

Otto Esko

Siirtoliipaistujen prosessorien käyttäminen FPGA-pohjaisissa järjestelmäpiireissä

Kandidaatintyö

Tarkastaja: Erno Salminen

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
OTTO ESKO: Siirtoliipaistujen prosessorien käyttäminen FPGA-pohjaisissa järjestelmäpiireissä
Kandidaatintyö, 50 sivua
Maaliskuu 2011
Pääaine: Digitaali- ja tietokonetekniikka
Tarkastaja: Erno Salminen
Avainsanat: järjestelmäsuunnittelu, sovelluskohtainen prosessori, räätälöitävä prosessori, siirtoliipaistu prosessoriarkkitehtuuri

Digitaalisten järjestelmien monimutkaistuminen ja alati kasvava transistoritiheys asettavat haasteita suunnittelutyölle. Tarve suunnitteluntyön tuottavuuden parantamiselle on ohjannut suunnittelua korkeammalle abstraktiotasolle. Järjestelmäsuunnittelussa suunnittelijan rakennuspalikoina ovat logiikkaporttien ja rekistereiden sijaan kokonaiset komponenttilohkot, jotka itsessään toteuttavan jonkin tietyn toiminnallisuuden. Komponenttilohkoja yhdistellään toisiinsa erilaisten väylien avulla, jolloin saadaan koottua kokonaisia järjestelmiä.

FPGA-piirit ovat uudelleenohjelmoitavia logiikkapiirejä, jotka tarjoavat mielenkiintoisen alustan järjestelmäsuunnittelulle. Uudelleenohjelmoitavuuden ansiosta FPGA-piireillä voidaan nopeasti testailla erilaisia järjestelmiä laitteistolla.

Järjestelmäsuunnittelu ei kuitenkaan ratkaise ongelmaa, miten suorituskykyisiä komponenttilohkoja voitaisiin toteuttaa helposti ja nopeasti. Tässä kandidaatintyössä esitetään ratkaisuksi sovelluskohtaiseksi räätälöitävien prosessorien käyttämistä komponenttilohkoina. Prosessorin avulla saadaan käyttöön ohjelmoitavuuden tuomat edut ja räätälöinnin avulla tavoitellaan suorituskykyä.

Työssä esitetään, miten räätälöitävä prosessori voidaan liittää järjestelmäsuunnitteluvuohon käyttäen esimerkkinä Koskivuota. Työssä automatisoitiin räätälöitävän prosessorin kääriminen järjestelmäsuunnitteluvuohon yhteensopivaksi komponenttilohkoksi. Lisäksi työssä osoitettiin räätälöitävän prosessorin yhteensopivuus Koskivuohon suunnitteluesimerkin avulla.

SISÄLLYS

TERMIT JA SYMBOLIT

ADF	Architecture Definition File
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
CPI	Cycles Per Instruction
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FU	Function Unit
GCU	Global Control Unit
GPP	General Purpose Processor
HDB	Hardware Database
HIBI	Heterogeneous IP Block Interconnection
HW	Hardware
IDF	Implementation Description File
ILP	Instruction Level Parallelism
IP	Intellectual Property
JTAG	Joint Test Action Group
LIMM	Long Immediate
LUT	Look-Up Table
MIMO	Multiple-Input and Multiple-Output
MMIO	Memory Mapped Input/Output
N2H2	Nios II to HIBI versio 2
NOP	No Operation
NRE	Non-Recurrent Engineering
OSEd	Operation Set Editor
PIG	Program Image Generator
ProGe	Processor Generator
RAM	Random Access Memory
RF	Register File
RISC	Reduced Instruction-Set Computer
RTL	Register Transfer Level
SIMM	Short Immediate
SRAM	Static Random Access Memory
SoC	System-on-Chip
SW	Software

TCE	TTA-based Codesign Environment
TLP	Thread Level Parallelism
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word
XML	eXtensible Markup Language

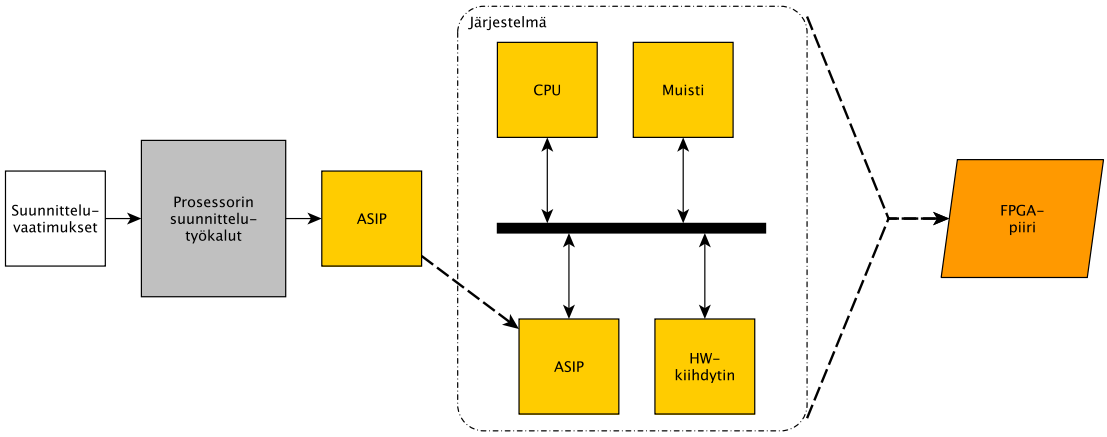
1. JOHDANTO

Järjestelmäpiirit (engl. System on Chip, SoC) ovat viime vuosikymmenien aikana yleistyneet digitaalisissa ja sulautetuissa järjestelmissä. Osaltaan tämä johtuu siitä, että vuosi vuodelta järjestelmien monimutkaisuus kasvaa ja se asettaa haasteita suunnittelutyölle. Jotta näihin haasteisiin voitaisiin vastata, suunnittelutyön tuottavuutta pitää lisätä. Tämä on käytännössä johtanut korkeamman abstraktiotason, järjestelmätason, käyttämiseen suunnittelussa. Siinä missä ennen suunnittelijan peruskomponentteja olivat logiikkaportit ja rekisterit, järjestelmäsuunnittelussa käytetään muun muassa kokonaisia komponenttilohkoja, prosessoreja ja muisteja, joita yhdistellään toisiinsa erilaisten väylien avulla. [1]

Yksi järjestelmäsuunnittelun tarjoamista suunnittelutyön tuottavuutta parantavista kulmakivistä on komponenttien uudelleenkäytettävyys. Komponentti suunnitellaan, toteutetaan ja varmennetaan kerran, mutta sitä voidaan hyödyntää useassa eri järjestelmässä, jolloin järjestelmän toteuttaminen ja varmentaminen nopeutuu. Uudelleenkäytettävyys on luonut myös uudenlaiset markkinat, jossa yritykset kauppaavat kehittämiään komponentteja. Tällaisia komponentteja kutsutaan usein IP-lohkoiksi (engl. Intellectual Property). [1] [2]

Osaltaan myös uudelleenohjelmoitavien logiikkapiirien, kuten FPGA-piirien (Field Programmable Gate Array), yleistyminen on helpottanut järjestelmäpiirien kehittämistä. FPGA:ta voidaan hyödyntää järjestelmien prototyyppien testaamiseen ennen kuin järjestelmästä valmistetaan ASIC-piirejä. Toisaalta FPGA-piirien kehittyminen ja ASIC-piirien korkeat kertakustannukset (engl. Non-Recurrent Engineering costs, NRE) ovat mahdollistaneet FPGA-piirien käyttämisen myös lopputuotteissa. Erityisesti pienivolyymisissa tuotteissa FPGA:n käyttäminen ASIC:n sijaan saattaa tulla merkittävästi halvemmaksi. [3] [4]

Järjestelmäsuunnittelussa on kuitenkin eräs perustavanlaatuinen pulma: mistä käytettävät komponentit tulevat? Komponenttien toteuttaminen laitteistolla on usein hidasta ja sen myötä kallista puuhaa, joten asialle olisi toivottavaa löytää jokin vaihtoehtoinen ratkaisu. Erityisesti FPGA-piireille on tarjolla useita niin sanottuja pehmytydinprosessoreita (engl. soft core processor), joita voidaan hyödyntää järjestelmäsuunnittelussa [5]. Ohjelmoitavan prosessorin käyttäminen mahdollistaa sen, että varsinaisen toiminnallisuuden voi toteuttaa ohjelmallisesti laitteiston sijaan. Ohjelmakoodin toteuttaminen onkin usein halvempaa ja huomattavasti nopeampaa kuin

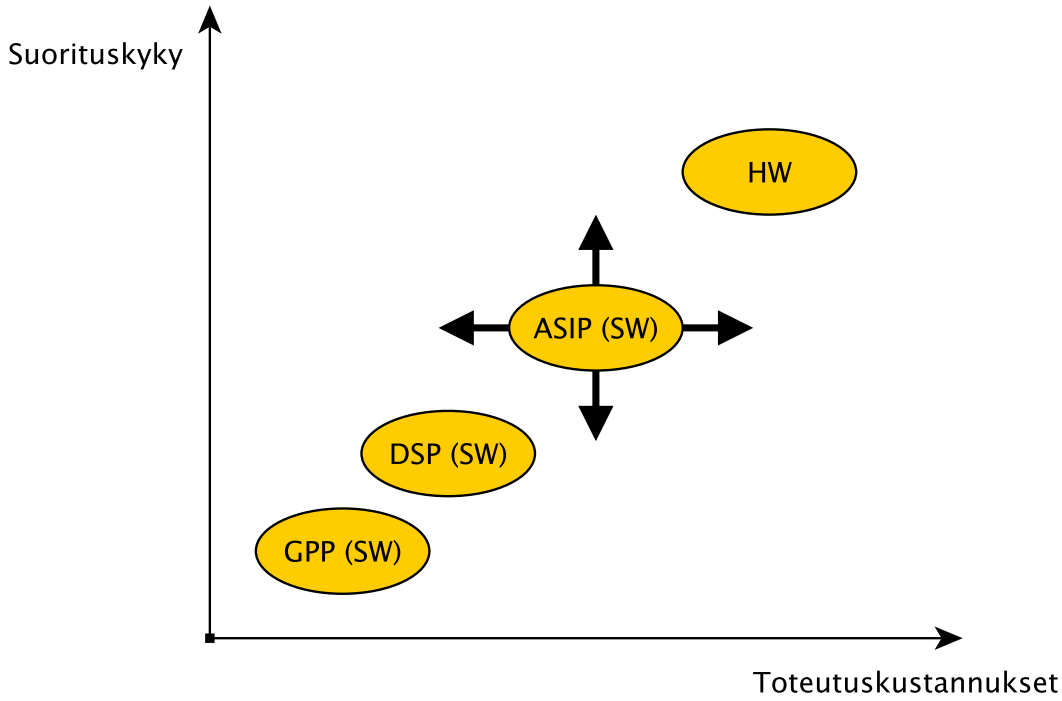


Kuva 1.1: Korkean tason kuvaus, miten suunnitteluvaatimusten mukaisesti räätälöity sovelluskohtainen prosessori (ASIP) saadaan osaksi FPGA:lle toteutettavaa järjestelmää.

saman toiminnallisuuden toteuttaminen suoraan laitteistolla. Varjopuolena on kuitenkin se, että ohjelmallinen toteutus häviää usein laitteistototeutukselle niin suoritusnopeudessa kuin energiankulutuksessa.

Tässä kandidaatintyössä esitetään ongelman ratkaisuksi sovelluskohtaiseksi räätälöitävien prosessoreiden eli sovelluskohtaisten prosessoreiden (engl. Application Specific Instruction-set Processor, ASIP) hyödyntämistä järjestelmäsuunnittelun komponenttien toteuttamisessa. Kuva 1.1 esittää pähkinänkuoressa, miten tämä ratkaisumalli toimii. Räätälöinnin lähtökohtana on tinkiä prosessorin yleiskäyttöisyydestä ja muokata prosessori mahdollisimman hyvin yhden sovelluksen, tai jossain tapauksessa tietyn sovellujoukon, suorittamista varten. Sovelluskohtaisuudella pyritään kiilaamaan suorituskyvyssä yleiskäyttöisen prosessorin ja laitteistototeutuksen väliin, kuten kuva 1.2 esittää. Tavoitteena on päästä mahdollisimman lähelle laitteistototeutuksen suorituskykyä säilyttäen ohjelmoitavuuden tuomat hyödyt. Lisäksi suorituskyky ja toteutuskustannuksia on joissain tapauksissa mahdollista skaalata räätälöintityön määrällä. Toisin sanoen, panostamalla enemmän tuotantoresursseja räätälöintiin, voidaan saavuttaa parempi suorituskyky.

Työn rakenne on seuraava. Kappaleessa 2 tehdään lyhyt katsaus tarjolla oleviin pehmytydinprosessoreihin ja esitellään työssä käytettävä siirtoliipaistu prosessoriarkkitehtuuri. Kappaleessa 3 kuvataan siirtoliipaistujen prosessorien suunnittelu- ja varmennusvuo. Neljännessä kappaleessa kerrotaan, miten siirtoliipaistu prosessori voidaan liittää järjestelmäsuunnitteluvuohon. Kappaleessa 5 osoitetaan suunnitteluesimerkin avulla, että siirtoliipaistun prosessorin liittäminen järjestelmään toimii. Viimeinen kappale on työn yhteenveto.



Kuva 1.2: Periaatteellinen kaavio ohjelmallisten toteutuksien (SW, software) ja laitteistototeutuksen (HW, hardware) eroavaisuuksista tehokkuuden ja toteutuskustannuksien välillä.

2. RÄÄTÄLÖITÄVÄT PROSESSORIARKKITEHTUURIT

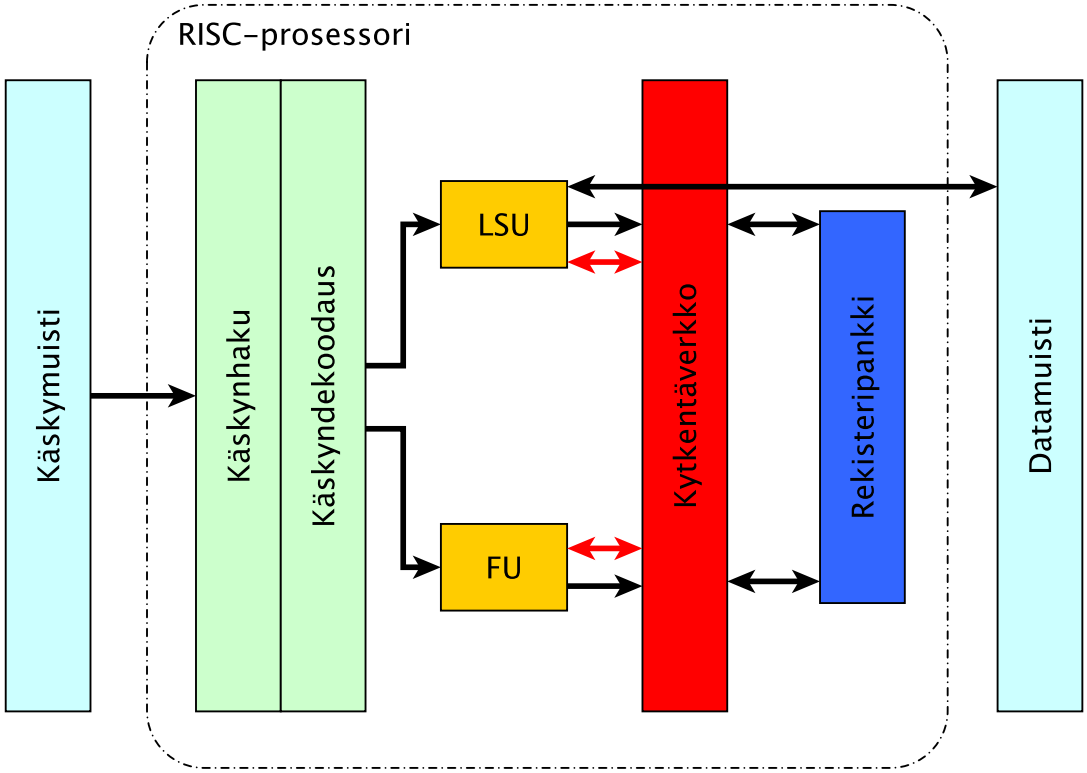
Tässä kappaleessa käydään läpi FPGA-piireille suunnattua prosessoritarjontaa ja näiden prosessorien räätälöintimahdollisuuksia. Lisäksi kappaleessa esitellään tässä työssä käytettävä, räätälöitävä prosessoriarkkitehtuuri ja verrataan sitä muihin prosessoriarkkitehtuureihin.

2.1 Nykytilanne

Skaalari-RISC (engl. Reduced Instruction Set Computer) arkkitehtuurille on tyypillistä, että prosessori suorittaa käskyjä liukuhihnoitetusti yksi käsky kerrallaan. Käskynsuoritus on jaettu useisiin lyheisiin vaiheisiin, joka mahdollistaa korkeamman kellotaajuuden käyttämisen verrattuna siihen, että käsky suoritettaisiin kokonaan yhden kellojakson aikana. Esimerkiksi perinteisessä RISC:ssä on viisi vaihetta, jotka ovat käskyn haku, käskyn dekoodaus, suoritus, muistinkäsittely ja takaisinkirjoitus. Liukuhihnoituksessa kaikkia näitä vaiheita hyödynnetään samanaikaisesti suorittamalla useita käskyjä lomitetusti. Tällöin prosessori kykynee aloittamaan uuden käskyn suorittamisen jokaisella kellojaksolla. Monivaiheisessa käskynsuorituksessa liukuhihnoituksen hyödyntäminen ei siis lyhennä yhden käskyn suorittamiseen kuluvaa aikaa, mutta käskyjen lomittaminen pienentään perättäisten käskyjen suorittamiseen kuluvaa kokonaisaikaa. Suorituskykyä voidaankin mitata *kellojaksoa per käsky*-lukemalla (engl. clock cycles per instruction, CPI), joka kuvaa keskimääräistä käskyn suorittamiseen kuluvaa kellojaksomäärää. Liukuhihnoituksen avulla *CPI*-lukema lähestyy yhtä.Kuvassa 2.1 on esitetty esimerkki skalaari-RISC-arkkitehtuurista. [6]

Käskytason rinnakkaisuus (engl. Instruction Level Parallelism, ILP) määrittelee, kuinka montaa käskyä jostakin sekventiaalisesta käskyvirrasta olisi mahdollista suorittaa yhtä aikaa. Esimerkiksi jos kaksi peräkkäistä yhteenlaskukäskyä operoi täysin eri operandeilla ja kirjoittaa tulokset eri rekistereihin, käskyt on mahdollista suorittaa rinnakkain. Jos taas yhteenlaskukäskyn toinen operandi riippuu edellisen käskyn tuloksesta, käskyillä on keksinäinen datariippuvuus, eikä niitä voida suorittaa täysin samaan aikaan. Liukuhihnoituksen hyöty perustuu osin käskytasonrinnakkaisuuteen. Jos peräkkäisillä käskyillä ei ole riippuvuuksia, ne voidaan suorittaa lomitetusti ilman, että liukuhihnaa tarvitse pysäyttää. [6]

Pehmytydin on prosessoriydin, joka syntesoidaan ohjelmoitaville logiikkapiireil-



Kuva 2.1: Esimerkkikuva skalaari-RISC-prosessoriarkkitehtuurista. Arkkitehtuurissa on on yksi laskentayksiikkö (engl. Function Unit, FU). Tämä FU on aritmeettislooginenyksikkö (engl. Arithmetic Logic Unit, ALU), jolla suoritetaan lasku-, loogiset- ja vertailuoperaatiot. Tämä lisäksi arkkitehtuurissa on datamuistiyksikkö (engl. Load Store Unit, LSU), jonka avulla käsitellään datamuistia.

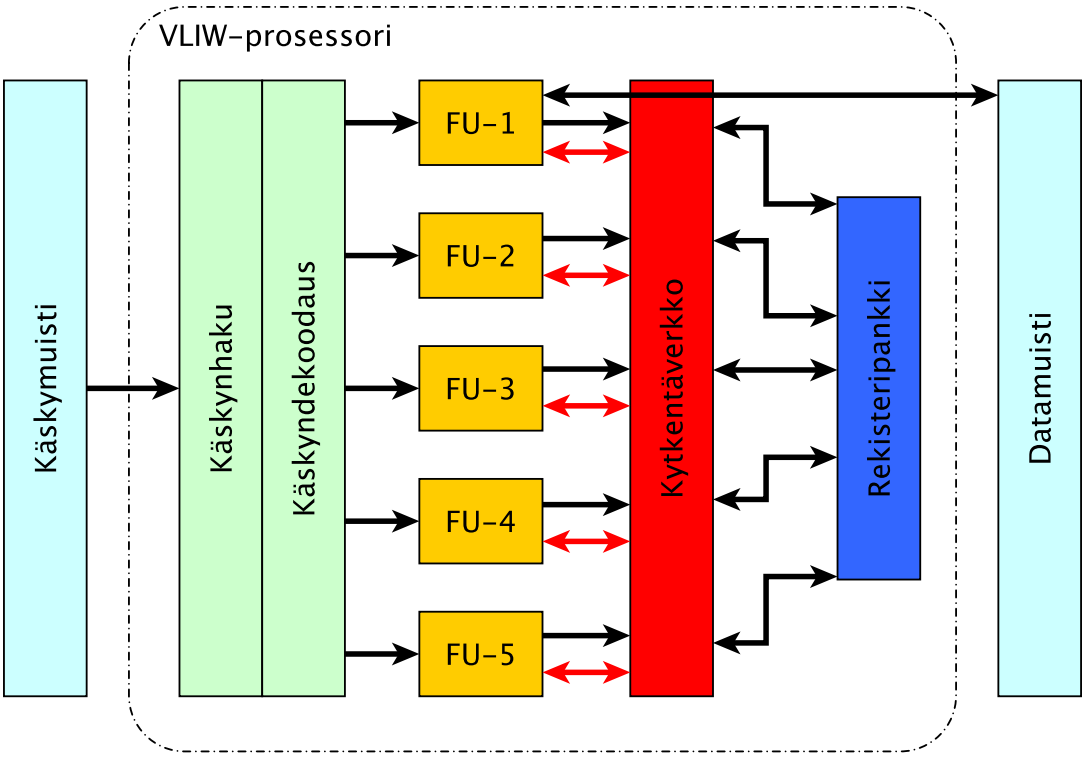
le käyttäen piirillä olevia logiikkaelementtejä [5]. FPGA-valmistajat, kuten Altera ja Xilinx, tarjoavat omille FPGA-piireillensä suunnattuja pehmytydinprosessoreita [7] [8]. Tämän lisäksi saatavilla on myös kolmannen osapuolen tarjoamia pehmytytimiä, joista osa, kuten OpenRisc 1200 [9], on julkaistu avoimena lähdekoodina. Ohjelmavoitavuuden ansiosta pehmytytimet yleensä tarjoavat suunnittelijalle mahdollisuuden muokata prosessorin ominaisuuksia, kuten lisätä arkkitehtuuriin raudalla toteutettu kertoja ja jakaja, tai muuttaa välimuistiasetuksia. Jotkut pehmytytimet, kuten Nios II antaa suunnittelijalle mahdollisuuden jopa käskykantalaajennukseen [7]. Näitä pehmytytimiä yhdistää se, että ne ovat kaikki skalaari-RISC:jä [5] [10]. Osittaisesta räätälöitävyydestä huolimatta näiden pehmytytimien suorituskyvyn skaalautuvuus ei välttämättä ole riittävä, sillä skalaari-RISC:n kyky hyödyntää käskytason rinnakkaisuutta on hyvin rajallinen, koska näillä arkkitehtuureilla *CPI* on aina suurempi tai yhtä suuri kuin yksi [6]. Patterson & Hennessy [6] mainitsevat kirjassaan kaksi päätapaa, miten prosessoriarkkitehtuurissa voidaan hyödyntää käskytason rinnakkaisuutta paremmin: kasvattamalla liukuhih-

nan syvyyttä tai lisäämällä suoritusyksiköiden määrää. Jälkimmäisellä vaihtoehdolla olisi mahdollista päästä alle yhden *CPI*-lukemaan, koska yhden kellojakson aikana voitaisiin suorittaa useampi kuin yksi käsky. Tämä kuitenkin vaatisi prosessoriarkkitehtuurityypin muuttamista, eikä mikään edellä mainituista pehmytytimistä tue sellaista. Liukuhihnan syvyyteen sen sijaan voi joissain määrin vaikuttaa. Esimerkiksi Nios II on tarjolla kolmena eri versiona: *economy*, jossa ei ole lainkaan liukuhihnaa, *standard*, jossa on 5-vaiheinen liukuhihna ja *fast*, jossa on 6-vaiheinen liukuhihna [7]. Liukuhihnan syventämisen avulla voidaan saavuttaa korkeampi kellotaajuus, joka oletettavasti parantaa suorituskykyä, mutta sen avulla ei voida päästä alle yhden *CPI*-lukemaan [6].

Resurssien hyötykäyttöä voidaan joissain tapauksissa parantaa lisäämällä useita prosessoriytimiä, mutta todellisen lisäsuorituskyvyn saavuttaminen vaatisi myös suoritettavan sovelluksen muokkaamista monisäikeiseksi, jolloin sovellus pystyy hyödyntämään useaa prosessoriydintä. Tällöin on kyse säietason rinnakkaisuuden hyödyntämisestä (engl. Thread Level Parallelism, TLP), jolloin jokaisen ydin suorittaa omaa säiettään. Käytännössä sovelluksen muokkaminen tällaiseksi on usein työlästä ja virhealtista, eikä suorituskykykään useimmiten kasva lineaarisesti ytimien määrän kasvaessa sovelluksissa esiintyvien riippuvuuksien ja suorituksen synkronoinnin tarpeen vuoksi [11]. Tästä syystä olisi houkuttelevampaa kasvattaa suorituskykyä käskytason rinnakkaisuutta hyödyntämällä, koska se ei tavallisesti vaadi suunnittelijalta suuria, työläitä muutoksia sovellukseen, toisin kuin säietason rinnakkaisuuden hyödyntäminen [6].

2.2 Kohti räätälöitävämpää pehmytydinarkkitehtuuria

Jotta arkkitehtuurin räätälöinnistä saataisiin enemmän hyötyä, arkkitehtuurin pitäisi pystyä myös hyödyntämään käskytason rinnakkaisuutta. VLIW (Very Long Instruction Word) eli erittäin pitkän käskysananarkkitehtuuri on tunnettu siitä, että se kykenee käyttämään hyväksi käskytason rinnakkaisuutta. Kuten kuvasta 2.2 ilmenee, tällaisissa arkkitehtuureissa on useita rinnakkaisia laskentayksiköitä, joissa voidaan suorittaa operaatioita rinnakkain. Yksi VLIW käsky koostuukin useasta rinnakkain suoritettavasta operaatiosta. Nimensä mukaisesti käskysanasta voi tällöin muodustua hyvinkin pitkä. VLIW:lle on lisäksi ominaista, että rinnakkain suoritettavat operaatiot määritellään jo käännösaikana. Täten arkkitehtuurissa ei tarvitse olla monimutkaista laitteistolla toteutettua käskyjen uudelleenjärjestelylogiikkaa, jolloin arkkitehtuurin laitteistototeutus yksinkertaistuu huomattavasti. Toisaalta tämä asettaa kääntäjän tärkeään rooliin suorituskyvyn kannalta. Jos kääntäjä ei kykyne löytämään tai pysty uudelleenjärjestelemään operaatioita siten, että niitä voidaan suorittaa samaan aikaan, rinnakkaisista laskentayksiköistä ei saada lisähyötä skalaari-RISC:iin nähden. Viime kädessä suorituskyky riippuu myös sovelluskoodis-



Kuva 2.2: Esimerkkikuva VLIW prosessoriarkkitehtuurista, jossa on 5 rinnakkaista laskentayksikköä. Laskentayksikkö on lyhennetty termillä FU (engl. Function Unit).

ta, sillä jos sovelluksessa ei juurikaan ole käskytason rinnakkaisuutta, ei paraskaan kääntäjä voi asialle mitään. [11]

VLIW:n potentiaalista suorituskykyä voidaan kasvattaa lisäämällä arkkitehtuuriin enemmän laskentayksiköitä, jolloin on mahdollista suorittaa yhä useampia operaatioita samanaikaisesti. Laskentayksiköiden lisääminen tuo kuitenkin melko nopeasti esille perinteisen VLIW-arkkitehtuurin pullonkaulat, joista ensimmäinen on rekisteripankki. Corporaal esittää, että VLIW:ssä, jossa on N kappaletta laskentayksiköitä, joilla on kaksi sisäänmenoporttia ja yksi ulostuloportti, tarvitsee keskitetyssä rekisteripankissa olla $3N$ kappaletta portteja, joista $2N$ kappaletta on lukuportteja ja N kappaletta kirjoitusportteja [12]. Tarve selittyy sillä, että pahimmassa tapauksessa jokaisen laskentayksikön pitää lukea tai kirjoittaa rekisteripankkia samaalla kellojaksolla. Lisäksi Corporaal esittää, että rekisteripankin kompleksisuus kasvaa vähintään lineaarisesti porttivaatimusten seuraksena [12]. Tutkimustulos osoittaa, että rekisteripankin kasvava kompleksisuus voi alkaa rajoittamaan VLIW prosessorin maksimikellotaajuutta FPGA-piirillä [13].

Toinen pullonkaula löytyy laskentayksiköiden välisestä rekisteriohikytkentäverkosta (engl. register bypassing network). Tämän ohitkytkentäverkon avulla voidaan

siirtää tuloksia laskentayksiköiden ulostuloista laskentayksiköiden sisäänmenoihin, jolloin seuraava operaatio voi käyttää laskentatulosta ilman rekisteristä lukemista. Corporaalin mukaan rekisteriohikytkentäverkon kompleksisuus kasvaa neliöllisesti laskentayksiköiden määrään nähden, jolloin ohikytkentäverkon vaatimat resurssit kasvavat nopeasti. [12]

Näiden pullonkaulojen vuoksi VLIW:n laskentayksiköiden määrän skaalattavuus on rajallinen. Pahimmassa tapauksessa laskentayksiköiden lisäämisellä voi olla negatiivinen vaikutus sovelluksen suoritusaikaan, jos suorituksen kesto kellojaksoissa mitattuna ei pienene yhtä paljon suhteessa kellotaajuden laskuun.

2.3 Siirtoliipaistu prosessoriarkkitehtuuri

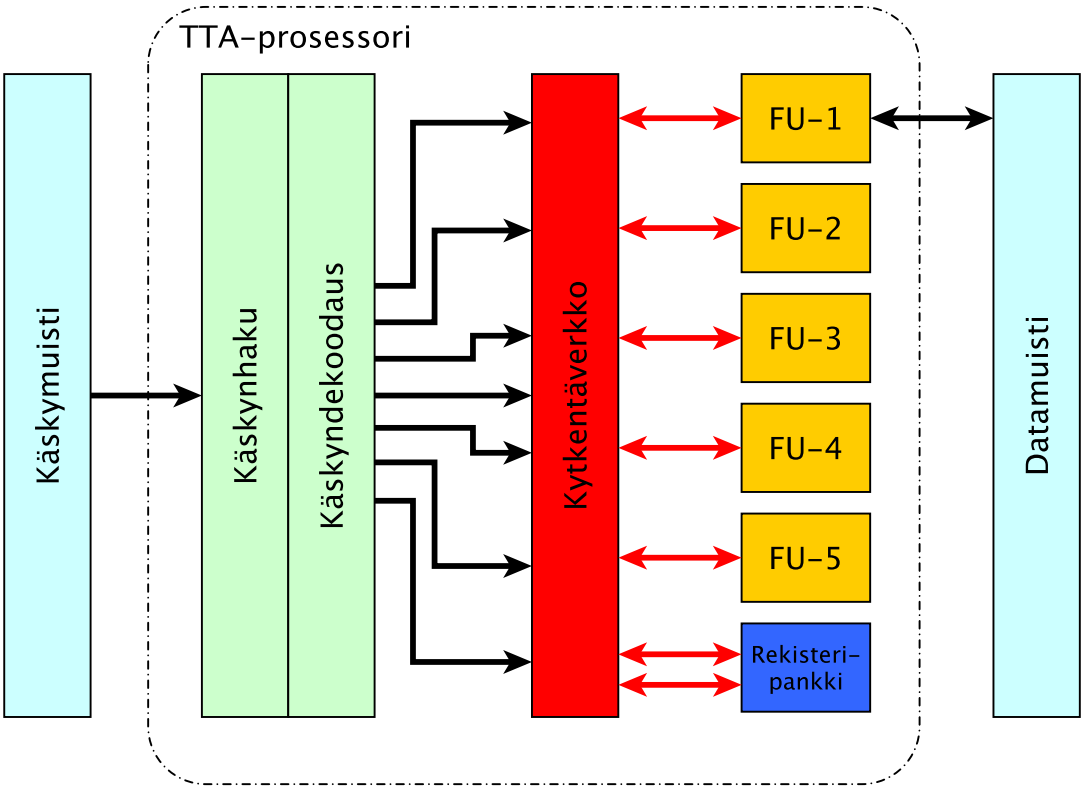
Siirtoliipaistu prosessoriarkkitehtuuri (engl. Transport Triggered Architecture, TTA) [12] on yksi ratkaisu VLIW-arkkitehtuurille ominaisiin pullonkauloihin. TTA on ennenkaikkea prosessorin suunnittelumalli (engl. template), joka avulla on mahdollista luoda sovelluskohtaiseksi räätälöitäviä prosessoreita. Seuraavissa aliluvuissa kuvataan TTA:n ominaisuuksia laitteiston ja ohjelmoinnin näkökulmista.

2.3.1 Kytkentäverkko

TTA on luotu VLIW:n pohjalta muuttamalla perinteisen operaatiopohjaisen prosessorin ohjelmointiparadigma siirtopohjaiseksi, jonka avulla pyritään kehittämään ratkaisu VLIW:n ongelmakohtiin. Perinteisessä operaatiopohjaisessa ohjelmointiparadigmassa käsky määrittelee suoritettavan operaation ja sen operandit, mutta TTA:ssa käsky määritteleekin vain operandisiirrot laskentayksiköiden ja rekisteripankkien välillä. Operaatiot taas suoritetaan näiden operandisiirtojen sivuvaikutuksena. Prosessoriarkkitehtuurin näkökulmasta tämä näkyy siten, että käsky ohjaa laskentayksiköiden välistä kytkentäverkkoa, kuten kuva 2.3 havainnollistaa, eikä suoraan laskentayksiköitä, kuten VLIW:n tapauksessa (kuva 2.2).

Kytkentäverkon ohjaaminen käskyllä tarkoittaa puolestaan sitä, että laskentayksiköiden ja rekisteripankkien väliset kytkennät ovat näkyvissä ohjelmoijalle. TTA:ta kutsutaankin tämän vuoksi niin sanotuksi avoimen datapolun arkkitehtuuriksi (engl. exposed datapath architecture). Operandisiirtojen ohjelmoinnissa voidaan käyttää vain olemassa olevia kytkentöjä. TTA-prosessorin suunnittelija voi päättää kytkentöjen määrän ja täten kytkentäverkon kompleksisuus on hallittavissa.

Itse kytkentäverkko muodostuu siirtoväylistä (engl. transport bus) ja töpseleistä (engl. socket), jotka kytkevät laskentayksiköiden portit siirtoväyliin, kuten kuva 2.4 havainnollistaa. Yhden käskyn aikana on mahdollista siirtää jokaista siirtoväylää pitkin yksi operandi ulostulotöpselistä sisäänmenotöpseliin, olettaen, että molemmat töpselit on kytketty samaan siirtoväylään. Siirtoväylien lukumäärä siis määrittelee

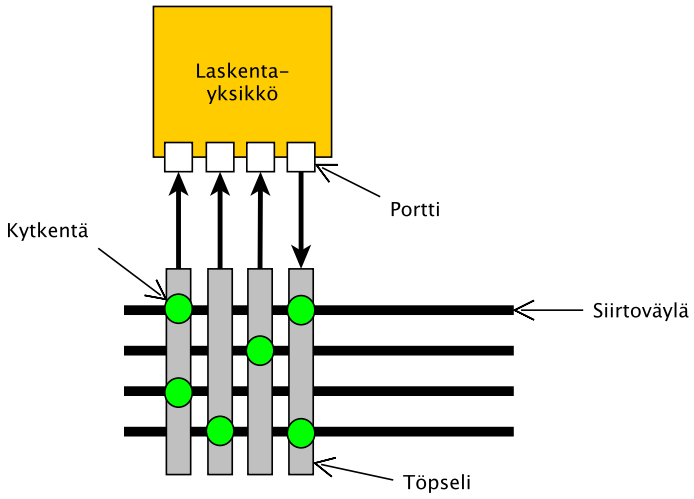


Kuva 2.3: Esimerkkikuva TTA prosessoriarkkitehtuurista, jossa on 5 laskentayksikköä. Laskentayksikkö on lyhennetty termillä FU (engl. Function Unit).

suoraan, kuinka monta rinnakkaista siirtoa voidaan maksimissaan tehdä yhden käskyn aikana. TTA:n käsky koostuu siis siirtokomennoista, joita on yhdessä käskyssä yhtä monta kuin siirtoväyliä. Jos jollain väylällä ei tehdä siirtoa, silloin käskyyn koodataan kyseisen siirtoväylän kohdalle ei operaatiota käsky (engl. no operation, NOP).

2.3.2 Laskentayksikkö ja rekisteripankki

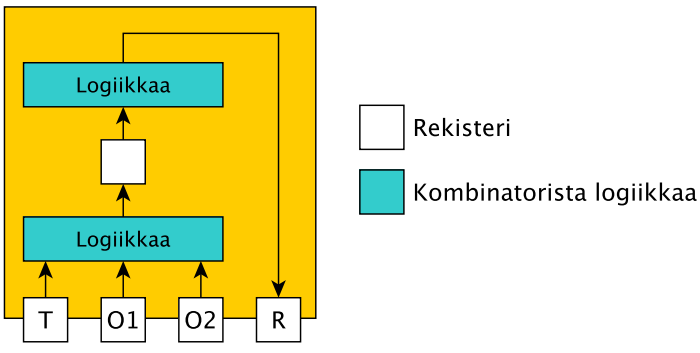
TTA prosessorissa on vain yksi käsky, joka on operandin siirto lähdetöpselistä kohdetöpseliin, ja varsinaiset operaatiot toteutetaan laskentayksiköissä. Laskentayksiköillä voi olla mielivaltainen määrä sisääntuloportteja, joihin operaatioiden sisäänmenooperandit kirjoitetaan, ja ulostuloportteja, joista operaatioiden tulokset luetaan. Lisäksi yksi laskentayksikkö voi toteuttaa yhden tai useamman operaation. Laskentayksikön rakennetta on kuvattu kuvassa 2.5, jossa sisäänmenoportit ovat vasemmalla ja ulostuloportti oikealla. Kun kyseisellä laskentayksiköllä halutaan laskea operaatio, sisäänmenoporttien operandirekistereihin $O1$ ja $O2$ siirretään operandit. Kolmannella sisäänmenorekisterillä, joka on merkitty T:llä, on erityistarkoitus. Tä-



Kuva 2.4: TTA prosessorin kytkentäverkon kytkennän rakenne. Suunnittelija määrää. kuinka moneen siirtoväylään laskentayksikkö kytketään.

mä on niin sanottu *liipaisurekisteri* (engl. trigger register), jolla operaatio käynnistetään. Eli kun tähän sisäänmenoon siirretään operandi, operaatio liipaistaan käyntiin. Tästä juontuu myös arkkitehtuurin nimi, *siirtoliipaistu arkkitehtuuri*.

Laskentayksikön operaatioilla on tavallisesti staattinen latenssi eli operaation ulostulo on luettavissa ulostulorekisteristä tietyn vakiokellojaksomäärän jälkeen operaation käynnistämisestä. Laskentayksikkö voi olla myös sisäisesti liukuhihnoitettu, kuten kuvan 2.5 tapauksessa. Operaatio on jaettu kahteen kombinatoriseen vaiheeseen ja jokaisen vaiheen välissä tulokset tallennetaan rekisteriin. Lopulta operaation tulos päätyy tulosrekisteriin R. Liukuhihnoituksen ansiosta on mahdollista käynnistää laskentayksikössä jokaisella käskyllä uusi operaatio. Lisäksi operaation tulos säi-



Kuva 2.5: Esimerkki TTA prosessorin laskentayksikön anatomiasta. Tämä laskentayksikkö ottaa 3 operandia sisään ja laskee niistä 2 kellojaksossa tuloksen rekisteriin R.

lyy laskentayksikön tulosrekisterissä niin kauan, kunnes uusi operaatio valmistuu. Laskentayksiköt ovat myös toisistaan riippumattomia, jolloin ne voivat suorittaa operaatioita rinnakkain.

TTA:n rekisteripankeissa on merkittävä ero VLIW:n (kuva 2.2) nähden, sillä TTA:ssa rekisteripankki on samanarvoinen laskentayksiköiden kanssa. Toisin sanoen rekisteripankki on liitetty kytkentäverkkoon samalla tavalla kuin laskentayksiköt, joten myös rekisteripankin kytkennät ovat näkyvissä ohjelmoijalle. Tämä poistaa VLIW:n vaatimuksen rekisteripankin luku- ja kirjoitusporttien määrän mitoittamisesta laskentayksiköiden lukumäärän mukaan.

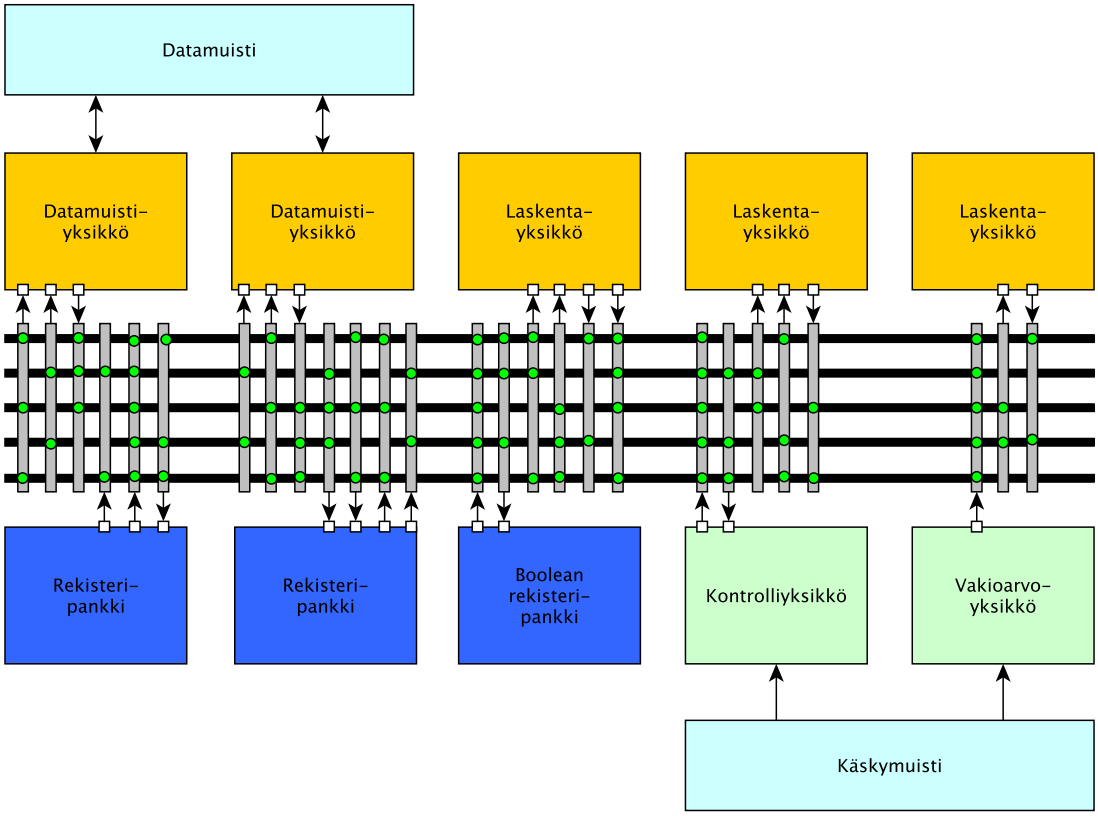
TTA-prosessorin ulkoiset liitynnät toteutetaan myös laskentayksiköiden avulla. Laskentayksiköllä voi olla arkkitehtuurin ulkoisia portteja, jotka eivät näy ohjelmoijalle. Tällaisten porttien avulla voidaan toteuttaa erikoislaskentayksikkö, jonka tarjoamilla operaatioilla voidaan vaikkapa vilkutella FPGA-laudan ledejä tai lukea nappuloiden ja kytkinten asentoja.

Ulkoisten porttien lisäksi toinen yleinen tapa liittyä prosessoriarkkitehtuurin ulkoisiin laitteisiin on käyttää niin sanottua muistikartoitettua liityntää (engl. memory mapped input/output, MMIO). Tällaiselle liitynnälle on tyypillistä, että ulkoisia komponentteja käytetään kirjoittamalla ja lukemalla komentorekistereitä, joille on annettu muistiosoite. Komennot siis kirjoitetaan prosessorin datamuistiyksikön (engl. load store unit) kautta, joka on normaalisti liitetty pelkästään datamuistiin.

2.3.3 Arkkitehtuuri tarkemmin

Tarkempi malli TTA prosessoriarkkitehtuurista on esitetty kuvassa 2.6, josta ilmenee myös arkkitehtuurin modulaarisuus. Uuden laskentayksikön tai rekisteripankin lisääminen onnistuu kytkemällä se töpseleiden avulla siirtoväyliin. Kuvasta 2.6 käy myös ilmi, miten prosessori on liitetty muistihin. Datamuistiyksikköä käytetään nimensä mukaisesti datamuistin käsittelyyn. Arkkitehtuurissa voi olla useita kappaleita datamuistiyksiköitä ja ne voivat käsitellä samoja tai eri datamuistiavaruuksia. Käskymuistiliityntä taas hoidetaan prosessorin kontrolliyksikön kautta (engl. global control unit, GCU). Kontrolliyksikkö hoitaa käskyjen hakemisen, dekoodauksen ja sitä myötä siirtojen käynnistämisen siirtoväylillä. Sen lisäksi kontrolliyksikkö tarjoaa hyppy- ja kutsuoperaatiot, joiden avulla ohjelman suoritusta voidaan haarauttaa.

Kuvassa 2.6 näkyy myös eräs TTA:n erikoisuus eli vakioarvoyksikkö (engl. immediate unit), joka on rekisteripankin kaltainen yksikkö. Tavallisesti vakioarvot koodataan käskyn lähdekenttään, joka TTA:n tapauksessa tarkoittaa, että siirtokäskyn lähdekentän lähdetöpselin tunnisteen tilalle kirjoitetaan suoraan vakioliteraali. Tämä kuitenkin tarkoittaa sitä, että siirron lähdekentän leveys pitää olla vähintään siirrettävän arvon levyinen. Esimerkiksi 32-bittinen vakioarvo vaatii vähintään 32-bittisen lähdekentän. Kun siirtoväyliä on useita, käskynleveys alkaa nopeasti paisua.



Kuva 2.6: Tarkempi kuvaus TTA prosessoriarkkitehtuurista.

Tämän vuoksi on yleistä, että siirtoväylien vakioarvoleveyksiä pienennetään, jolloin puhutaan niin sanotuista lyhyistä vakioarvoista (engl. short immediate, SIMM). Tällöin tarvitaan vakioarvoyksikköä pidempien vakioarvojen (engl. long immediate, LIMM) tukemiseen. Vakioarvoyksikköä käytettäessä pitkälle vakioarvolle tehdään kokonaan oma käskyformaatti, jonka avulla käskydekooderi voi kirjoittaa vakioliteraalin vakioarvoyksikköön. Tämän jälkeen vakioarvo voidaan lukea vakioarvoyksikön rekisteristä, jolloin lähdekenttien leveys voidaan pitää lyhyenä.

Totuusarvorekisterit ovat lisäksi ominaisia TTA:lle. Kuvassa 2.6 näkyy yksi totuusarvorekisteripankki, joka sisältää yksibittisiä rekistereitä. Näiden rekisterien avulla voidaan toteuttaa ehdollisia siirtoja. Esimerkiksi vertailuoperaation tulos voidaan kirjoittaa tällaiseen totuusarvorekisteriin ja seuraavalla käskyllä suoritetaan siirto sen mukaan, onko rekisterin arvo tosi vai epätosi. Jos ehto ei toteudu, siirtoa ei suoriteta, vaan se liiskataan (engl. squash) väylältä.

2.3.4 Ohjelmointimalli

Siirtopohjaisuuden vuoksi TTA prosessorin ohjelmointi eroaa perinteisen käskypohjaisen prosessorin ohjelmoinnista. Valoitetaan asiaa esimerkin avulla. Oletataan

RISC-pohjainen symbolinen konekieliformaatti (engl. assembly), jossa määritellään käsky, kohderekisteri ja operandit. Tällöin kahden rekisterin yhteenlasku $r3 = r1+r2$ voitaisiin tehdä komennolla:

```
add r3, r1, r2
```

TTA:n assembly-kielessä taas määritellään operandien siirrot rekistereiden ja laskentayksikön välillä. Yksi käskyrivi kuvaa yhden kellojakson aikana tapahtuvia operandisiirtoja. Täten operaation suoritus voitaisiin tehdä seuraavilla siirroilla:

```
RF.1 -> add.o1
RF.2 -> add.trigger
add.result -> RF.3
```

Tai jos käytettävissä olisi kaksi väylää, sisäänmeno-operandit voidaan siirtää rinnakkain:

```
RF.1 -> add.o1, RF.2 -> add.trigger
add.result -> RF.3
```

Assembly-koodista on myös nähtävissä operaatioiden latenssi. Esimerkin tapauksessa add-operaation latenssi on yhden kellojakson verran eli operaation tulos voidaan lukea liipaisua seuraavalla käskyllä. Jos taas add:n latenssi olisi kaksi kellojaksoa, pitää tuloksen lukemista ennen odottaa yksi kellojakso. Koodissa kolme pistettä kuvaa NOP-operaatiota, jolloin väylällä ei tapahtu siirtoja. Koodi on siis seuraavanlainen:

```
RF.1 -> add.o1, RF.2 -> add.trigger
...
add.result -> RF.3
```

Latenssin peittämiseen voidaan hyödyntää laskentayksiköiden liukuhihnoitusta. Esimerkiksi kahden peräkkäisen add-operaation, joilla on 2 kellojakson latenssi, laskeminen voitaisiin hoitaa seuraavasti:

```
RF.1 -> add.o1, RF.2 -> add.trigger
RF.3 -> add.o1, RF.4 -> add.trigger
add.result -> RF.3
add.result -> RF.5
```

Liukuhihnoituksen lisäksi voidaan hyödyntää muitakin optimointeja. Oletataan seuraavanlainen tilanne, jossa ensimmäisen operaation tulosta käytetään seuraavan operaation laskemisessa. RISC:n assembly-kielellä tilanne hoidataan vastaavasti:

```
add r3, r1, r2
xor r4, r3, r5
```

TTA:n tapauksessa voidaan hyödyntää niin sanottua ohjelmallista ohituskytkentää (engl. software bypassing) ensimmäisen operaation tuloksen siirtämisessä toisen operaation sisäänmenoon:

```
RF.1 -> add.o1, RF.2 -> add.trigger
add.result -> xor.o1, RF.5 -> xor.trigger
xor.result -> RF.4
```

Tällainen menettely voi merkittävästi vähentää rekisteripankin käyttötarvetta, jolloin rekisteripankin luku- ja kirjoitusporttien lukumäärä voidaan pitää pienenä. [14]

TTA tukee myös käyttäjän määrittelemiä erikoisoperaatioita (engl. custom operation). Erikoisoperaatioiden tapauksessa kääntäjän ei tarvitse tietää, mitä erikoisoperaatio tekeen, koska se on ohjelmoijalla vastuulla. Sen sijaan kääntäjä pystyy generoimaan erikoisoperaatiota käyttävää ohjelmakoodia, kunhan kääntäjä vain tietää erikoisoperaation sisäänmeno-operandien ja tulosoperandien lukumäärän ja operaation latenssin. Näiden tietojen avulla kääntäjä kykenee kirjoittamaan erikoisoperaatioiden sisäänmenosiirrot ja tietää, milloin operaation tulos voidaan aikaisintaan lukea ulostulosta. Lisäksi siirtopohjaisuus helpottaa monisisäänmenoisten ja moniulostuloisten (engl. Multiple-Input and Multiple-Output, MIMO) operaatioiden toteuttamista, koska kaikkia operandeja ei ole pakko siirtää yhden käskyn aikana.

2.4 Yhteenveto arkkitehtuureista

Esitellyistä arkkitehtuureista on koostettu yhteenvetotaulukko 2.1. Voidaan havaita, että skalaari-RISC-tyyppisten prosessoreiden, kuten Nios II ja MicroBlaze, ongelmana on heikko skaalautuvuus. Skalaari-RISC:ien yhdistäminen moniydin-prosessoriksi taas tuottaa ongelmia sovelluskehitykseen, koska suorituskyvyn parantaminen vaatii sovellukselta säietason rinnakkaistumista. VLIW-arkkitehtuuri pystyy hyödyntämään käskytason rinnakkaisuutta, mutta laskentayksiköiden määrän skaalaamista rajoittaa arkkitehtuuriin muodostuvat pullonkaulat rekisteripankin ja kytkentäverkon osalta. TTA puolestaan antaa prosessorisuunnittelijalle enemmän vapausasteita, jolloin pullonkaulojen muodostumista voidaan hallita.

Taulukko 2.1: Yhteenveto eri arkkitehtuutien ominaisuuksista. Taulukossa RISC tarkoittaa skalaari-RISC:iä.

Arkkitehtuuri	**Laskenta-yksiöitä**	**Ohjelmointi-paradigma**	**Rinnakkaisuus**	**Ongelma**
RISC	1	Operaatio	(ILP)	Ei skaalaudu.
Moniydin-RISC	ydinten määrä	Operaatio	TLP	Ohjelmointi hankalaa.
VLIW	skaalattava	Operaatio	ILP	Rekisteripankin ja kytkentäverkon pullonkaulat rajoittavat skaalautumista.
TTA	skaalattava	Operandi-siirto	ILP	Soveltuu huonosti kontrollipainotteisiin sovelluksiin.

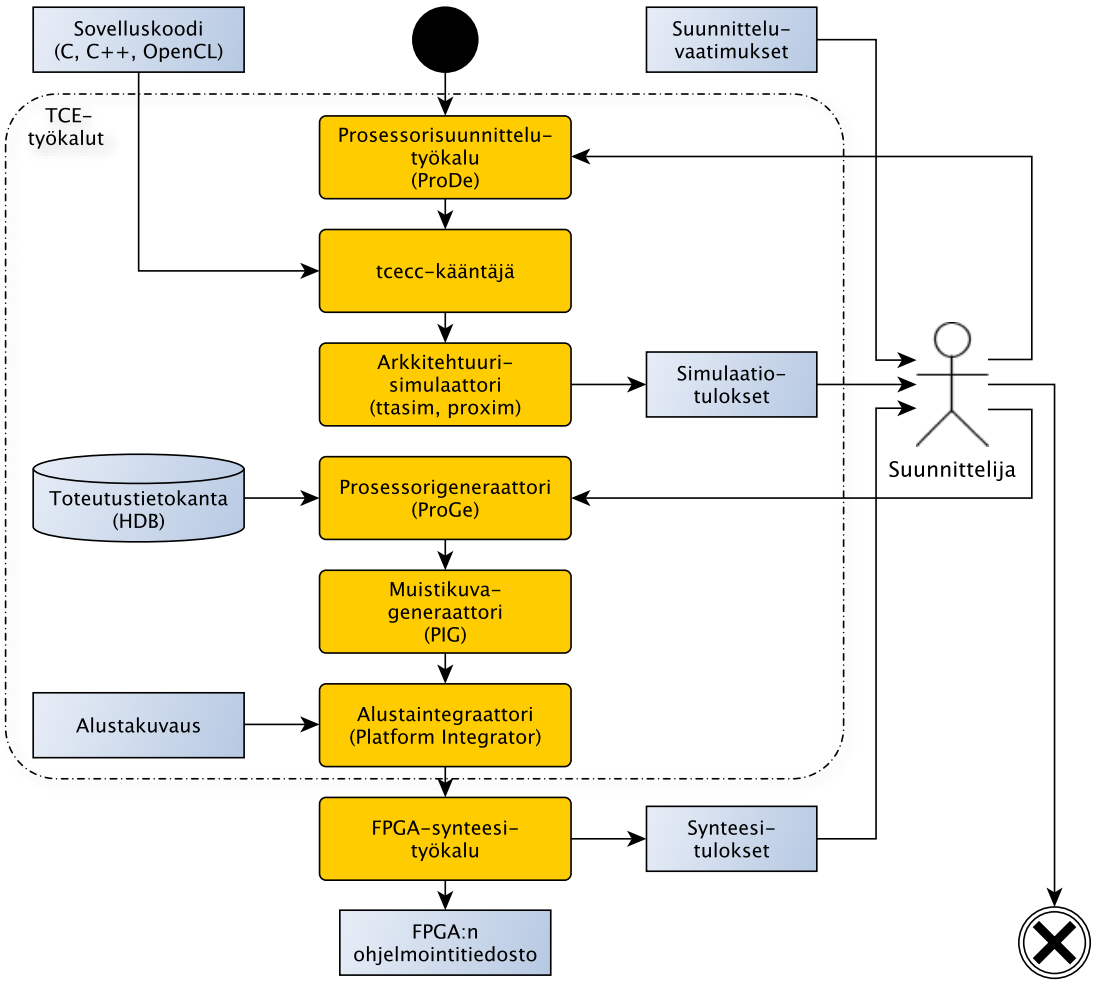
3. SIIRTOLIIPAISTUJEN PROSESSORIEN SUUNNITTELUVUO

Siirtoliipaistu prosessori sopii hyvin sovelluskohtaiseen käyttöön helpon räätälöitävyytensä ansiosta. Usein ovelluskohtainen prosessorin suunnittelun lähtökohtana on luoda sellainen arkkitehtuuri, jolla jotakin tiettyä sovellusta voidaan suorittaa riittävän nopeasti tai energiatehokkaasti. Suunnitteluprosessille asetetaan usein suoritusnopeuden lisäksi muitakin reunaehtoja, kuten kohdepiirin resurssien käyttö tai kellotaajuus. On hyvin epätodennäköistä, että ensimmäisellä yrityskerralla onnistuttaisiin luomaan sellainen prosessoriarkkitehtuuri, joka täyttäisi kaikki annetut vaatimukset. Tämän vuoksi TTA-prosessorien suunnittelu on iteratiivinen prosessi, kuten suunnitteluvuokuvasta 3.1 käy ilmi. Suunnitteluvuo on lyhyesti esitelty myös kirjoittamassani konferenssijulkaisussa [15].

Vuossa käytetään TTA-based Codesign Environmentia (TCE) eli TTA-prosessorien ja sovellusten rinnakkaiskehitysympäristöä. TCE-työkalujen avulla on mahdollista muokata TTA-prosessoreita yksityiskohtaisesti, kääntää ohjelmia TTA-prosessoreille sekä luoda prosessoritoteutuksen laitteistonkuvauskielillä. TCE:n tarjoamat tärkeimmät prosessorinmuokkauskohteet ovat laskentayksiköt, rekisteripankit ja siirtoväylät sekä näiden väliset kytkennät. Laskentayksiköiden määrää voidaan muokata lähes mielivaltaisesti, samoin kuin mitä operaatioita laskentayksikkö sisältää. Yksiköiden sisään- ja ulostuloporttien määrä on myös muokattavissa. Vastaavasti myös rekisteripankkien määrää voidaan vaihdella. Tämän lisäksi myös rekisteripankin rekisterien lukumäärää ja leveyttä sekä luku- ja kirjoitusporttien määrää voidaan muunnella. Siirtoväylien lukumäärä on myös valittavissa ja ennen kaikkea yksiköiden ja siirtoväylien välisiä kytkentöjä voidaan muokata. [16] [17]

3.1 Vuon vaiheet

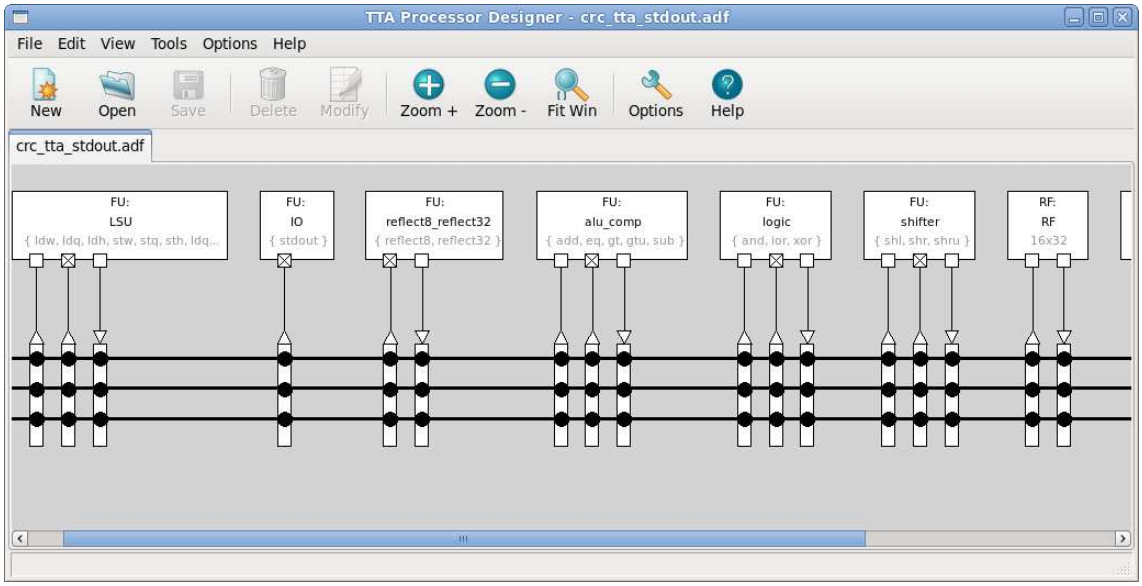
Suunnitteluvuon sisääntulona on suunnitteluvaatimukset, jotka suunniteltavan prosessorin tulisi täyttää, ja sovelluskoodi, jota suunniteltavalla prosessorilla halutaan suorittaa. Iterointi aloitetaan joko olemassa olevalla TTA-prosessoriarkkitehtuurikuvauksella tai luomalla kokonaan uusi arkkitehtuurikuvaus prosessorisuunnittelutyökalu ProDe:lla (Processor Designer). Arkkitehtuurikuvaus (Architecture Definition File, ADF) on XML-tiedosto (engl. eXtensible Markup Language), jossa kuvataan muun muassa prosessorin laskentayksiköt, rekisteripankit ja siirtoväylät sekä



Kuva 3.1: TTA-prosessorin suunnitteluvuo. Tässä työssä Alustaintegraattori on tärkeässä osassa, sillä sen avulla toteutetaan prosessorin yhteensopivuus kohdejärjestelmään.

niiden väliset kytkennät. Kuvassa 3.2 on ProDe-työkalun päänäkymä, jossa on esillä eräs TTA-prosessori. Kun aloitusarkkitehtuuri on valittu, ensimmäisenä vaiheena on kääntää sovellus kyseiselle arkkitehtuurille. Tämä tehdään tcecc-kääntäjällä, joka mukautuu automaattisesti ajonaikana kulloiseenkin arkkitehtuuriin [18]. Tämä on erityisen tärkeä ominaisuus suunnittelunopeuden kannalta, sillä perinteisesti arkkitehtuurin muuttuessa pitää toteuttaa kääntäjä uudelleen, jotta se osaisi kääntää ohjelman uudelle arkkitehtuurille. Käännöksen tuloksena saadaan ohjelmatiedosto TPEF (TTA Program Exchange Format).

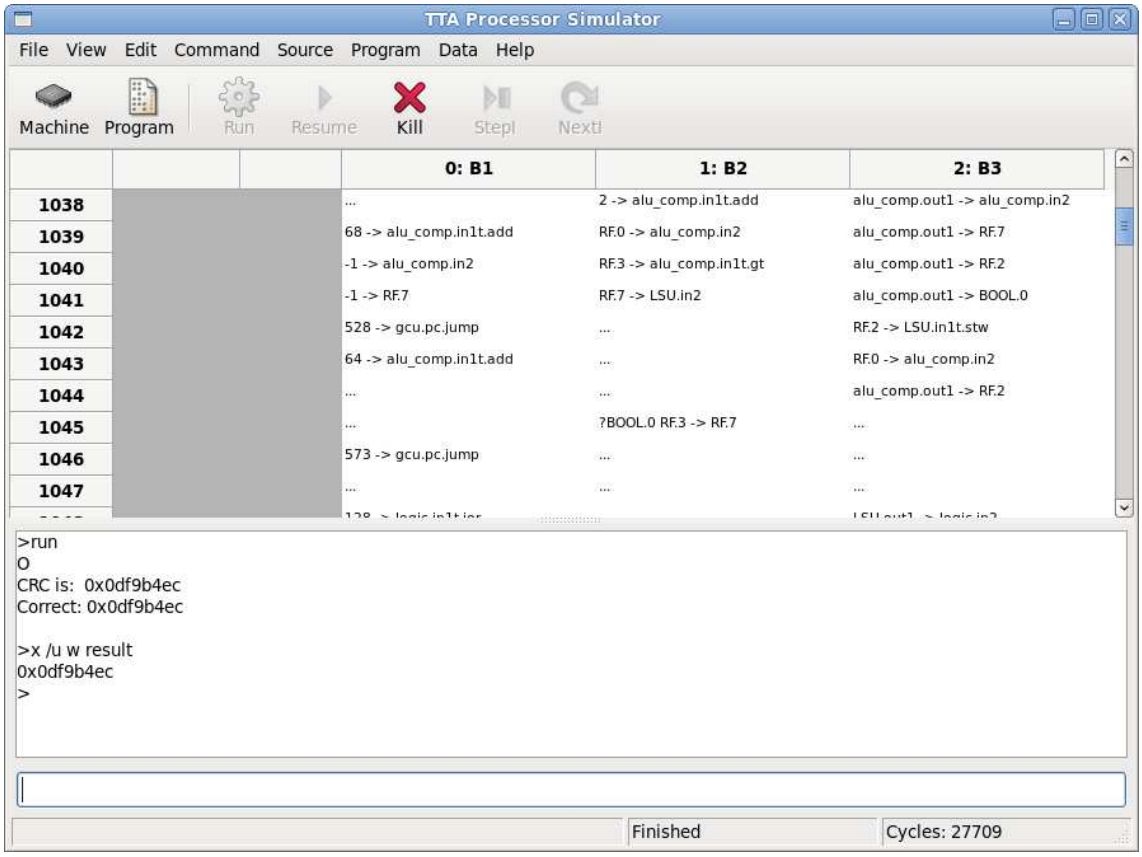
Kääntämisen jälkeen seuraava vaihe on sovelluksen simulointi, joka suoritetaan arkkitehtuurisimulaattorilla, joko komentorivipohjaisella ttasim:llä tai Proxim:lla, joka tarjoaa graafisen simulaattorikäyttöliittymän. Kuvassa 3.3 on esitetty kuvankaappaus graafisesta Proxim-arkkitehtuurisimulaattorista. Kuten kääntäjä, myös



Kuva 3.2: Kuvankaappaus TTA-prosessorien suunnittelutyökalu ProDe:sta.

arkkitehtuurisimulaattori mukautuu ajonaikaisesti käytettävään TTA-prosessoriarkkitehtuuriin [19]. Simulaatioajosta saadaan tuloksia, kuten suoritettujen kellojaksojen määrä ja prosessorin resurssien käyttöaste, joita suunnittelija tulkitsee. Näiden tulosten analysoinnin perusteella muokataan prosessoriarkkitehtuuria ProDe-työkalulla. Jos tuloksista esimerkiksi näkyy, että prosessorin siirtoväylien käyttöaste on korkea, uuden siirtoväylän lisääminen saattaisi pienentää suoritettavien kellojaksojen määrää, koska tällöin voidaan tehdä useampia siirtoja rinnakkain. Rekisterien, laskentayksiköiden ja eri operaatioiden käyttöasteesta voidaan tehdä vastaavanlaisia päätelmiä ja muokata arkkitehtuuria päätelmien mukaisesti. Muokkaamisen jälkeen aloitetaan uusi iteraatiokierros, jolloin uusista simulaatiotuloksista voidaan tutkia, miten aiemmin tehdyt muokkaukset vaikuttivat suorituskykyyn. On hyvä huomata, että tässä vaiheessa suorituskykyä ei voida vielä mitata ajassa, koska arkkitehtuurin kellotaajuus on vielä tuntematon, joten tuloksista nähdään vain muutoksien vaikutus kellojaksojen määrään. Jos suunnittelun reunaehdot määrittelevät jonkin tietyn kellotaajuuden, sallitun maksimikellojaksomäärän laskeminen on helppoa. Tällöin ei ole mielekästä jatkaa vuota ensimmäistä iteraatiosilmukkaa pidemmälle, ennen kuin kyseinen maksimikellojaksomäärä on alitettu. Tätä ensimmäistä iteraatiosilmukkaa on myös mahdollista automatisoida explorer-työkalun avulla. Työkalulle voidaan antaa esimerkiksi kellotaajuustavoite, johon se pyrkii pääsemään kasvattamalla arkkitehtuurin resursseja.

Kun arkkitehtuurin muokkaamisessa on päästy siihen pisteeseen, missä suorituksen kellojaksomäärä tyydyttää suunnittelijaa tai suunnittelija haluaa saada selville arkkitehtuurin maksimikellotaajuuden ja resurssien käytön, siirrytään vuossa eteen-



Kuva 3.3: Kuvankaappaus arkkitehtuurisimulaattori Proxim:sta

päin kohti prosessorin laitteistokuvauskielisen toteutuksen generointia. Ennen prosessorin generointia on kuitenkin valittava arkkitehtuurikuvauksen resursseille niitä vastaavat toteutukset. Tätä varten TCE:ssä on niin sanottuja laitteistotietokantoja (Hardware Database, HDB), joihin voidaan tallentaa laitteistonkuvauskielisiä toteutuksia laskentayksiköistä ja rekisteripankeista sekä niiden kustannustietoja. Jokainen HDB:ssä oleva toteutus on numeroitu yksilöllisesti, joten toteutuksien valinnassa yksinkertaisesti määritellään, mitä toteutusta mistäkin HDB-tiedostosta arkkitehtuurikuvauksen laskentayksiköt ja rekisteripankit vastaavat. Näiden lisäksi on mahdollista valita myös prosessorin kontrolliyksikön ja kytkentäverkon generointiliitännäinen (GCU/IC-plugin). Toteutuksien valinnat tallennetaan XML-muotoiseen toteutuskuvaustiedostoon (Implementation Definition File, IDF) myöhempää käyttöä varten.

Arkkitehtuuri- ja toteutuskuvauksien avulla TCE:n prosessorigeneraattori ProGe (Processor Generator) luo laitteistokuvauskielisen toteutuksen prosessorista. ProGe hakee laskentayksiköiden ja rekisteripankkien toteutukset HDB-tiedostoista ja käyttää GCU/IC-plugin:ia kontrolliyksikön luomiseen. Lopuksi GCU/IC-plugin luo yksiköitä ja rekisteripankkeja yhdistävän kytkentäverkon, joka koostuu arkkitehtuu-

rikuvauksessa määritellyistä siirtoväylistä ja niiden kytkennöistä.

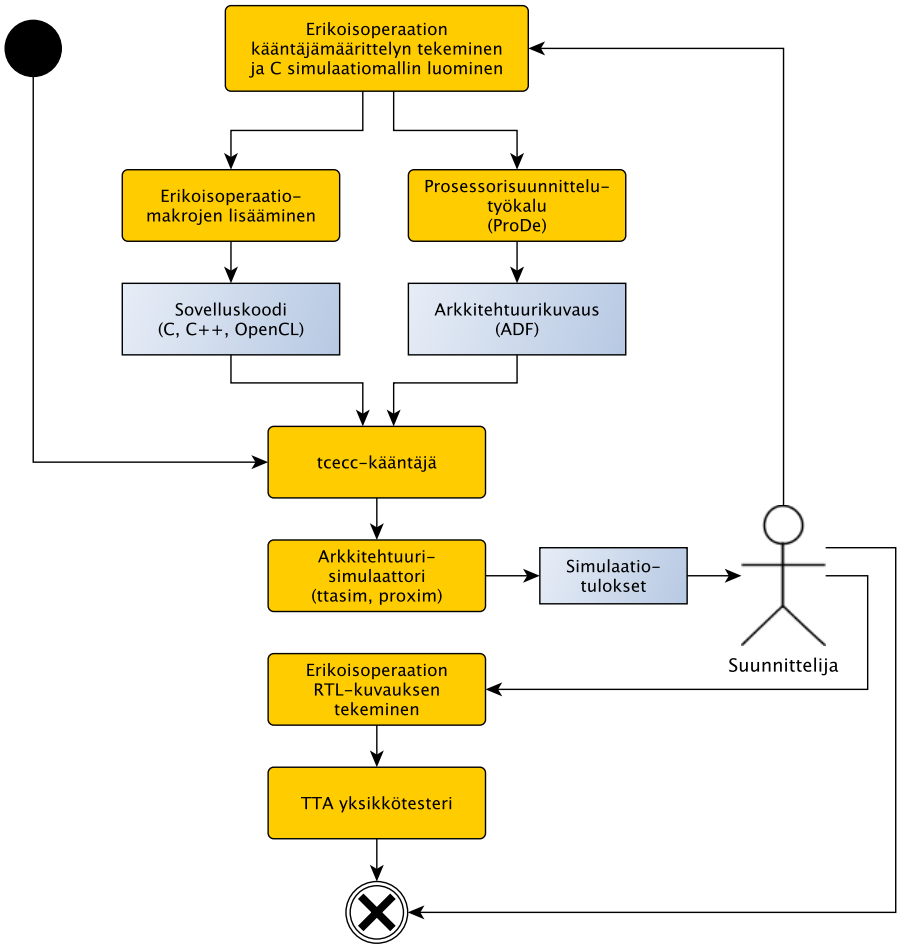
Prosessori tarvitsee myös suoritettavan ohjelman. Tätä varten vuon aiemmassa vaiheessa TPEF-muotoon käännetty ohjelmakoodi pitää muuntaa sellaiseen muotoon, jossa se voidaan siirtää muistikomponentille. Tämä onnistuu muistikuvageneraattori PIG:llä (Program Image Generator), joka erottelee TPEF:stä käsky- ja datamuistit ja generoi niistä muistikuvat (engl. image) suunnittelijan määrittelemillä formaateilla.

Tähän mennessä on generoitu vasta prosessoriytimen toteutus ja ohjelmaimaget. Jotta prosessoriytimen voisi ottaa oikeasti käyttöön, se pitää integroida kohdealustalle eli tässä tapauksessa kohteena olevalle FPGA-piirille. Integroinnin voi suorittaa manuaalisesti, mutta TCE tarjoaa Platform Integrator eli alustaintegraattorityökalun, joka hoitaa asian automatisoidusti. Työkalulla voidaan integroita TTA-prosessori joko itsenäisesti FPGA:lla ajettavaksi tai TTA-prosessori voidaan paketoida IP-lohkoksi, kuten tässä työssä esitellään. Platform Integrator:n tehtävänä on liittää prosessoriydin muistikomponentteihin ja tarpeen mukaan yhdistää ytimen signaaleja FPGA:n pinneihin. Tämän lisäksi työkalu voi luoda kolmannen osapuolen synteesityökaluihin projektitiedostot ja syntesointiskriptin, joiden avulla prosessorin syntesointi onnistuu vaivattomasti.

Kun suunnittelija suorittaa prosessorin syntesoinnin, tuloksena saadaan FPGA-ohjelmointitiedosto, jonka avulla prosessori voidaan ladata FPGA-piirille. Tämän lisäksi synteesityökalu ilmoittaa muun muassa prosessorin todellisen resurssienkäytön sekä kellotaajuuden, jonka avulla voidaan laskea sovelluksen suoritusaika. Saatuja tuloksia verrataan suunnitteluvaatimuksiin. Jos suunnittelija on tyytyväinen tuloksiin, vuo päättyy ja sovelluskohtainen prosessori on valmis. Jos taas tuloksissa on parantamisen varaa, suunnittelijalla on kaksi vaihtoehtoa. Ensimmäinen vaihtoehto on aloittaa jälkimmäinen iteraatiosilmukka uudelleen valitsemalla eri toteutukset arkkitehtuurikuvauksen komponenteille ja suorittaa uusi synteesi. Toinen vaihtoehto on palata ensimmäiseen iteraatiosilmukkaan ja muokata arkkitehtuurikuvausta. Vuota iteroidaan niin kauan, kunnes päästään haluttuun lopputulokseen. Huomionarvoista tässä suunnitteluvuossa on se, että suurin osa työstä tehdään arkkitehtuuritasolla, jolloin iterointi on nopeaa.

3.2 Käskykannan laajentaminen erikoisoperaatioilla

Arkkitehtuuritasolla tapahtuva suunnittelu mahdollistaa myös yksinkertaisen tavan testata suunnittelijan määrittelemiä erikoisoperaatioita. Kuva 3.4 esittää erikoisoperaatioiden suunnitteluvuota. Kuten prosessorinsuunnittelussakin, vuo alkaa ohjelmakoodin kääntämisestä nykyiselle arkkitehtuurikuvaukselle, jonka jälkeen seuraa simulaatio. Nyt simulaatiotuloksista pyritään tunnistamaan ohjelmakoodin hitaita, usein toistuvia kohtia, joita voitaisiin kiihdyttää erikoisoperaatiolla.



Kuva 3.4: Erikoisoperaatioiden suunnitteluvuo.

Kun erikoisoperaatiokandidaatti löytyy, aloitetaan erikoisoperaation luominen tekemällä operaatiolle kääntäjämääritelmä sekä C-kielinen simulaatiomalli OSEd-työkalun (Operation Set Editor) avulla. Operaation kääntäjämääritelmän luominen kuulostaa hankalalta tehtävältä, mutta sen avulla vain yksinkertaisesti kerrotaan kääntäjälle, kuinka monta sisäänmenoa ja ulostuloa operaatiolla on ja käsitelläänkö niitä etumerkillisinä vai -merkittöminä kokonaislukuina vai liukulukuina. Simulaatiomallin tekemisessä voi puolestaan hyödyntää sovelluksen alkuperäistä koodia, jota halutaan kiihdyttää. Esimerkiksi jos erikoisoperaation on tarkoitus korvata jokin sovelluksen funktio, simulaatiomallin voi määritellä funktion ohjelmakoodin avulla. Yleensä riittää, että parametrien ja paluuarvon välitys muutetaan siten, että sisääntulevat parametrit luetaan laskentayksikön sisääntuloporteista ja paluuarvot kirjoitetaan ulostuloportteihin.

Jotta uutta erikoisoperaatiota voitaisiin käyttää, arkkitehtuurikuvaukseen pitää lisätä vähintään yksi laskentayksikkö, joka toteuttaa tämän erikoisoperaation. Huo-

mionarvoista on se, että operaatioiden latenssi määritellään laskentayksikössä. Tässä vaiheessa erikoisoperaation latenssi ei ole vielä tiedossa, koska operaatiosta ei ole tehty laitteistonkuvauskielistä toteutusta, joten suunnittelijan täytyy tehdä valistunut arvaus. Toisaalta erikoisoperaation latenssin muuttamien laskentayksiköstä on vaivatonta ja suunnittelija voi halutessaan haarukoida, miten operaation latenssin vaihteleminen vaikuttaa suorituskykyyn.

Arkkitehtuurikuvauksen muokkaamisen lisäksi suunnittelijan pitää myös muuttaa ohjelmakoodia siten, että sovellus hyödyntää erikoisoperaatiota. Erikoisoperaatiota kutsutaan TCE-makrojen avulla. Esimerkiksi jos erikoisoperaatiot korvaavat jonkin sovelluksen funktion, suunnittelijan tulee korvata kaikki nämä funktiokutsut erikoisoperaation TCE-makrolla.

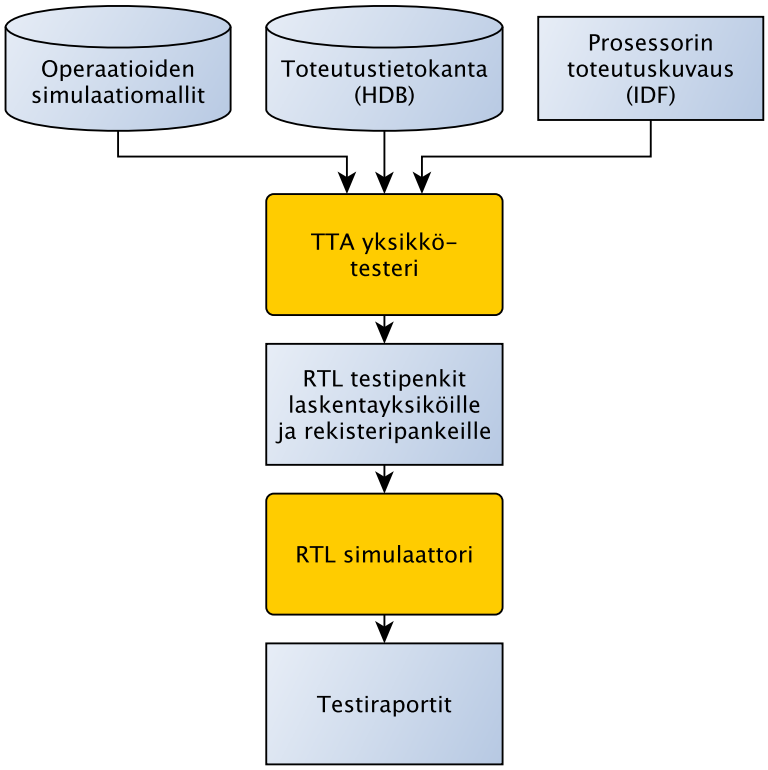
Muokkauksien jälkeen ohjelmakoodi käännetään vasta luodulle arkkitehtuurille ja simulaation jälkeen voidaan analysoida, miten erikoisoperaatio vaikutti suoritettujen kellojaksojen määrään. Jos tulokset eivät olleet halutunlaisia, suunnittelija voi kokeilla seuraavaa erikoisoperaatiokandidaattia ja aloittaa uuden iteraatiokierroksen. Vasta, kun suunnittelija löytää sellaisen erikoisoperaation, joka tuottaa merkittävän parannuksen, vuossa siirrytään eteenpäin operaation toteuttamiseen. Tässä vaiheessa suunnittelijan pitää tehdä erikoisoperaation sisältävälle laskentayksikölle laitteistokuvauskielinen toteutus, jotta prosessorigeneraattori pystyisi luomaan prosessoritoteutuksen. Tämä on ainoa kohta koko TCE-suunnitteluvuossa, jossa suunnittelijan pitää itse kirjoittaa laitteistonkuvauskielistä koodia. Kun erikoisoperaation toteutus on valmis, se lisätään HDB-tietokantaan, jolloin yksikköä on mahdollista uudelleenkäyttää tulevissa suunnitteluprojekteissa. Viimeinen vaihe tässä vuossa on erikoisoperaatiototeutuksen testaaminen yksikkötesterillä, jonka tarkoituksena on varmentaa, että erikoisoperaation laitteistokuvauskielinen toteutus vastaa erikoisoperaation C-kielistä simulaatiomallia. Yksikkötesteriä käsitellään tarkemmin luvussa 3.3.1.

3.3 Varmennusvuo

On äärimmäisen tärkeää, että suunnittelun tulos voidaan varmentaa. Tässä luvussa kuvataan pääpiirteittäin TTA-prosessorien varmentaminen TCE-ympäristössä. Varmentaminen on hyvä aloittaa mahdollisimman matalalta tasolta ja edetä askel kerrallaan kohti kokonaista järjestelmään, jotta virheet voidaan löytää mahdollisimman aikaisessa vaiheessa. Toisaalta tällainen menettelytapa auttaa myös rajaamaan virheen esiintymispaikkaa, jolloin virheen paikallistaminen helpottuu.

3.3.1 Yksikkötestaus

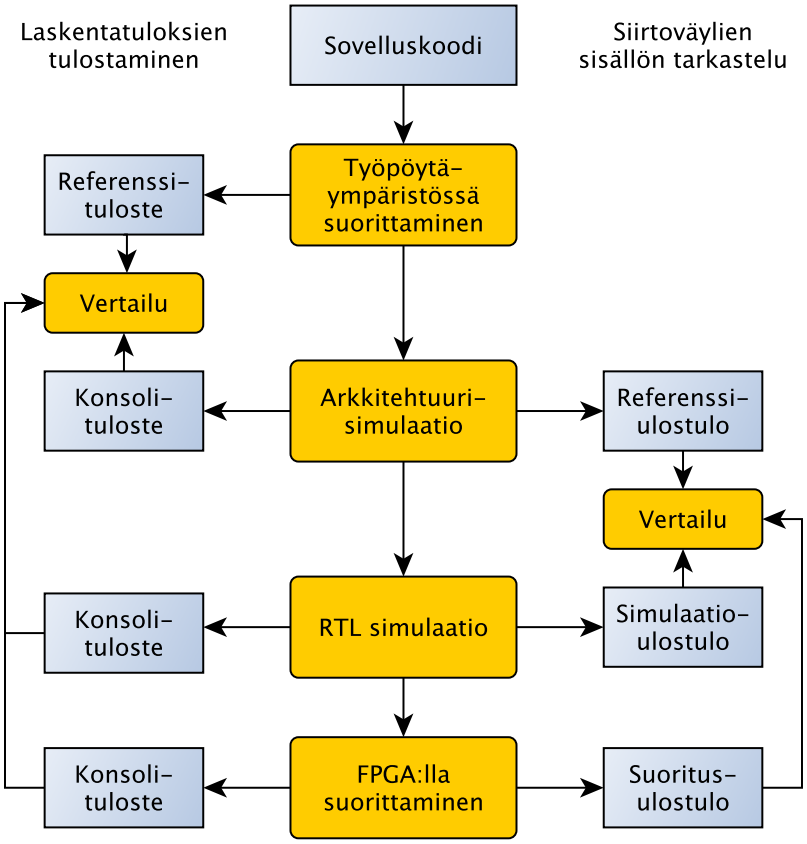
Prosessorin testauksen matalin taso alkaa lasketayksiköiden ja rekisteripankkien testaamisesta. Koska prosessorin suunnittelu tehdään yksinomaan arkkitehtuuritasol-



Kuva 3.5: TTA yksikkötesterin toimintaperiaate.

la, on ensisijaisen tärkeää varmentaa, että laskentayksiköiden ja rekisteripankkien laitteistonkuvauskieliset toteutukset vastaavat niiden arkkitehtuuritason simulaatiomalleja. Muutoin prosessorin arkkitehtuurimalli käyttäytyy samoilla syötteillä eri tavalla kuin generoitu prosessoritoteutus.

Yksikkötesterin toimintaperiaate on esitetty kuvassa 3.5. Se hakee yksi kerrallaan toteutuskuvaustiedosto IDF:ssä määritellyt laskentayksiköt ja rekisteripankit toteutustietokannoista ja generoi jokaiselle testattavalle yksikölle sisäänmenosyötteitä. Samalla yksikkötesteri tuottaa syötteitä vastaavat referenssiulostulot operaatioiden simulaatiomallien avulla. Tämän jälkeen yksikkötesteri generoi laitteistokuvauskielisen testipenkin kullekin laskentayksikön ja rekisteripankin toteutukselle. Testipenkki syöttää laskentayksikölle samat sisäänmenosyötteet kuin simulaatiomalleillekin ja vertaa laskentayksikön ulostulovasteita simulaatiomalleilta saatuihin ulostuloihin. Testipenkkiä ajetaan kolmannen osapuolen RTL-simulaattorilla, kuten kaupallisella Modelsimillä [20] tai avoimeen lähdekoodiin perustuvalla ghdl-simulaattorilla [21]. Jos simulaation ulostulo ja referenssiulostulo eroavat toisistaan, on helppo todeta, että yksikön toteutus ei vastaa simulaatiomallia. Jos taas ulostulot olivat yhtäläiset, voidaan tehdä oletus, että toteutus vastaa simulaatiomallia ainakin jollakin tarkkuudella. Täydellisen varmuuden saavuttaminen vaatisi sitä, että laskentayksikön



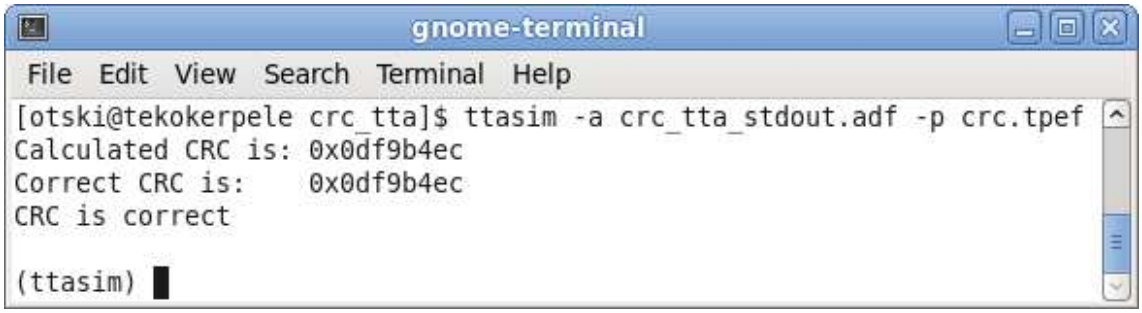
Kuva 3.6: Kaksi menetelmää TTA-prosessorin ja sovelluksen rinnakkaisvarmennukseen.

toteutusta testattaisiin kaikilla mahdollisilla sisääntulokombinaatioilla, tai käyttämällä jotain muuta menetelmää.

3.3.2 Prosessorin ja sovelluksen rinnakkaisvarmennus

Prosessorin suunnittelun ja muokkaamisen ohella on tärkeää pitää huoli siitä, että sovellus toimii oikein jokaisella iteraatiokierroksella ja jokaisella suunnittelutasolla. Kuva 3.6 esittää karkeasti kaksi tapaa, miten prosessorin ja sovelluksen toimintaa voidaan varmentaa eri suunnittelutasoilla. Nämä suunnittelutasot ovat tärkeässä osassa mahdollisten virheiden paikantamisessa. Esimerkiksi työpöytäympäristön ja arkkitehtuurisimulaation tuloksien poikkeavuus voi kieliä siitä, että sovelluskoodi ei ole standardien mukaista tai sovelluskoodille mahdollisesti tehdyt siirrettävyysmuokkaukset ovat virheellisiä. Harvinaisessa tapauksessa ongelmat saattavat johtua myös virheellisesti toimivasta kääntäjästä.

Vastaavasti arkkitehtuurisimulaation ja RTL-simulaation poikkeavuuksista voidaan tehdä johtopäätöksiä. Ennen RTL-simulaatiota tulisi kuitenkin suorittaa prosessorin yksikkötestaus, jotta virheellisistä laskentayksiköistä johtuvat ongelmat saa-



Kuva 3.7: Esimerkki ohjelmallisesta varmennuksesta. Ohjelmakoodissa on tulosteet laskenta- ja referenssitulokselle, jotka tulostetaan arkkitehtuurisimulaattorin pääikkunaan.

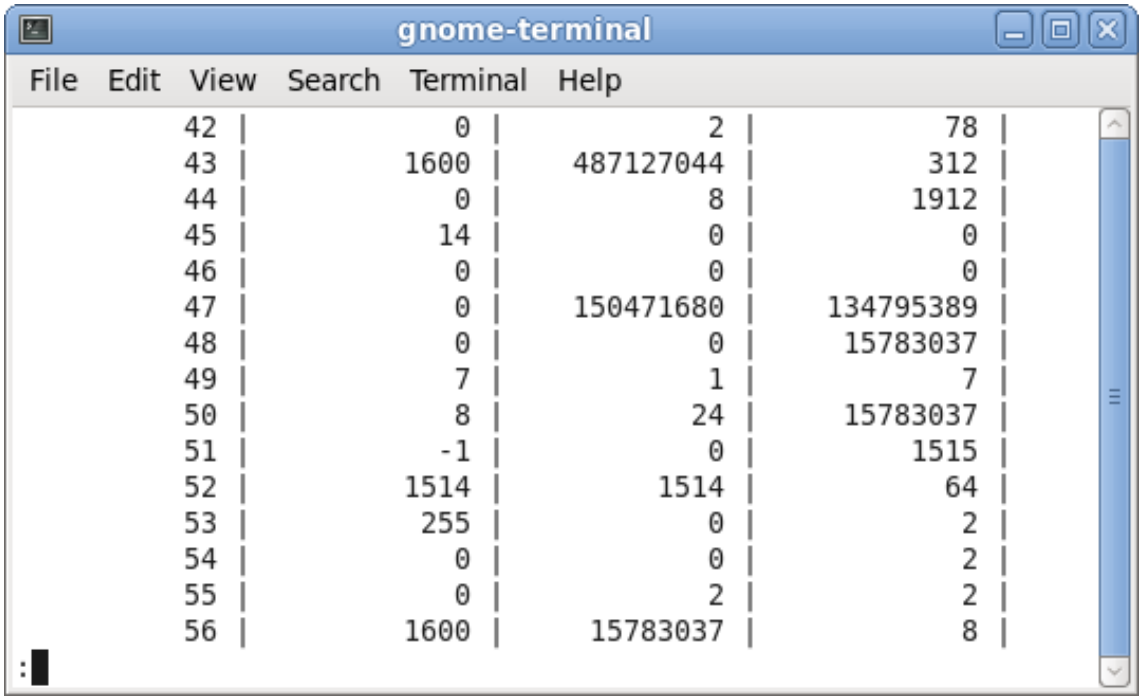
taisiin eliminoitua. Jos laskentayksiköt ovat toimivia, mutta RTL-simulaatio poikkeaa referenssistä, virheet saattavat johtua komponenttien integrointiongelmasta.

Viimeisellä tasolla ilmenevät virheet voivat myös johtua monesta asiasta. Ongelman ratkonnan voi aloittaa tarkistamalla, että prosessorin integrointi kohde-FPGA:lle on tehty oikein. Esimerkiksi vääränlaiset pinnikytkennät voivat aiheuttaa odottamattomia ongelmia. Myös väärän kellotaajuuden käyttäminen tai ajoitusvirheet voivat olla syypäänä.

Ohjelmallinen varmennus

Ohjelmallinen varmennus on esitetty kuvan 3.6 vasemmassa puoliskossa. Ohjelmallinen varmennus perustuu siihen, että ohjelmakoodiin lisätään tulosteita, jotka tulostavat esimerkiksi laskentatuloksia tai tilatietoja, joiden avulla voidaan seurata, käyttäytyykö sovellus oletetulla tavalla ja säilyykö käyttäytyminen muuttumattomana suunnittelun edetessä. Tulosteiden tuottamiseen voidaan käyttää C-kielen standardifunktioita, kuten `printf()`:ää tai `puts()`:ia. Oletusarvoisesti tulostusfunktiot käyttävät TCE:ssä `STDOUT`-nimistä operaatiota yhden merkin tulostamiseen. Prosessoriarkkitehtuuriin pitää siis lisätä erikoislaskentayksikkö, joka toteuttaa `STDOUT`-operaation, jotta tulostusfunktiota voisi käyttää. Kuvassa 3.7 on esimerkki ohjelmallisesta varmennuksesta.

Tulosteita käyttämällä sovelluksen referenssiulostulo voidaan generoida työpöytäympäristössä. Arkkitehtuurisimulaatiossa `STDOUT`-operaation simulaatiomalli tulostaa oletuksena merkit simulaattorin pääikkunaan, josta ne voidaan tallentaa tiedostoon vertailua varten. RTL-simulaatiovaiheessa taas `STDOUT`-operaation toteutuksessa voidaan hyödyntää syntesoitumattomia rakenteita, kuten VHDL:n textio-kirjastoa, jonka avulla tulostettavat merkit voidaan ohjata tiedostoon. Tulostusteiden saaminen FPGA:lta sen sijaan saattaa vaatia alustariippuvaisia toteutuksia, mutta yleensä FPGA-laudat tarjoavat joko UART- tai JTAG UART-liitynnän. Näiden liityntöjen avulla tulostetut merkit saadaan ohjattua terminaaliohjelmaan, ja



Kuva 3.8: Esimerkki siirtoväylien tarkastelusta. Kuvassa on arkkitehtuurisimulaatiosta saatu tuloste siirtoväylien sisällöstä. Vasemman puoleisessa sarakkeessa on ilmaistu ajanhetki kellojaksoina ja seuraavat kolme saraketta näyttävät siirtoväylien sisällöt kyseisellä kellojaksolla.

siitä edelleen tiedostoon. Esimerkiksi Alteran työkalujen ja FPGA-lautojen kanssa voi hyödyntää Nios II:lle suunnattua JTAG UART IP-lohkoa [22], jonka avulla tulosteet saadaan luettua nios2-terminal:n kautta.

Prosessorin siirtoväylien tarkasteluun perustuva varmennus

Tulosteiden lisäksi varmennusta voidaan tehdä tarkemmalla tasolla näytteistämällä prosessorin siirtoväylien sisältöä (engl. bus trace) jokaisella kellojaksolla. Tämä on esitetty kuvan 3.6 oikeassa puoliskossa. Siirtoväyliä voidaan kuitenkin tarkastella vasta arkkitehtuurisimulaatiossa, joten tätä varmennustapaa ei voida käyttää työpöytäympäristösuorituksen vertailemiseen. Siirtoväylien sisältö tallentuu arkkitehtuurisimulaation aikana tiedostoon, kun simulaattorin *bus trace*-asetus on päällä. Kuvassa 3.8 on esimerkki, miltä siirtoväylien tarkastelun tuloksena saatu tiedosto näyttää.

Vastaavasti siirtoväylien sisältö saadaan kaapattua RTL-simulaatiossa, kun kontrolliyksikköön lisätään komponentti, joka tulostaa siirtoväylien sisällön tekstitiedostoon. Tämän tiedonkeruuyksikön lisääminen onnistuu helposti laittamalla asetus päälle prosessorin generoinnin yhteydessä, eikä suunnittelijan tarvitse lisätä sitä käsin. Tiedonkeruuyksikkö käyttää hyväksi VHDL:n textio-pakettia, jolloin se ei ole

syntesoitavissa. FPGA-suorituksen aikana siirtoväylien sisältöä voisi kerätä esimerkiksi jonkinlaisen JTAG-komponentin avulla, mutta tällaista komponenttia ei ole vielä toteutettu.

3.4 Yhteenveto

TCE-suunnitteluympäristön tärkeimmät ominaisuudet on koottu taulukkoon 3.1.

Taulukko 3.1: Yhteenveto TCE-suunnitteluympäristöstä

Kohta	**Selite**
Nimi	TTA-Based Codesign Environment, TCE
Tarkoitus	Sovelluskohtaisten TTA-prosessorien ja sovellusten rinnakkaissuunnitteluohjelmisto.
Erityistä	Suunnittelutyö tehdään arkkitehtuuritasolla, jolloin suunnittelu on nopeaa. Suunnittelutyökalut mukautuvat arkkitehtuuriin ajonaikaisesti.
Tärkeimmät työkalut	tcecc-kääntäjä, ProDe-prosessorisuunnittelutyökalu, ttasim/proxim-arkkitehtuurisimulaattori, ProGe-prosessorigeneraattori ja PIG-muistikuvageneraattori
Kääntäjä	Hyödyntää LLVM:ää (Clang) [23]
Kielituki	C (myös kokeellinen C++ ja OpenCL tuki)
Viimeisin julkaistu versio	1.3 (10.11.2010) [17]
Lisenssi	MIT, LGPL (osa laitteistototeutuksista)
Tuetut käyttöjärjestelmät	Virallisesti Linux, kokeellisesti myös Mac OS X
Ohjelmiston toteutuskieli	C++
Tuetut laitteistokuvauskielet	VHDL

4. INTEGROINTI KOSKI-VUOHON

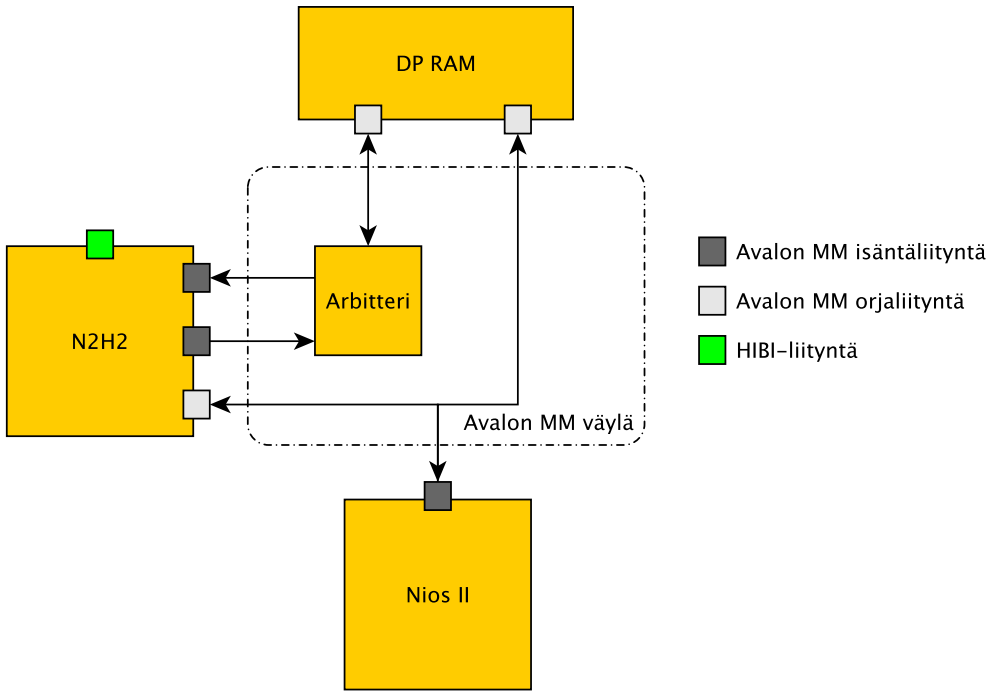
Koski on Tampereen Teknillisessä Yliopistossa kehitetty järjestelmäpiirien suunnitteluvuo, joka hyödyntää UML-mallinnusta (Unified Modeling Language) järjestelmän suunnitteluvaatimuksien hallinnassa. Lisäksi Koskivuon työkalut, kuten useat kaupalliset järjestelmäsuunnittelutyökalut, hyödyntävät komponenteista kuvattua metadataa järjestelmäpiirin eri osien integroinnissa. [24]

IP-XACT on IP-lohkojen metadatan kuvaukseen tehty kieliriippumaton standardi [25]. Sen kehittämisestä vastasi SPIRIT Consortium, joka on nykyisin osa Accelleraa. Varsinainen metadata tallennetaan XML-muotoisesti IP-XACT-tiedostoon. IP-XACT:n versio 1.2 esimerkiksi määrittelee omat metadataobjektit komponenteille, jotka ovat usein IP-lohkoja tai prosessoreja, ja väylämäärittelyille [25]. Komponentin metadatassa voidaan määritellä, että komponentissa on liityntä johonkin tiettyyn väyläarkkitehtuuriin ja lisäksi tieto siitä, mitä väylän signaalia mikäkin komponentin rajapinnassa oleva portti vastaa. Näiden metatietojen avulla järjestelmäsuunnittelutyökalu, kuten Koskivuon tapauksessa Kactus-työkalu, pystyy automaatisoidusti integroimaan komponentteja toisiinsa [26].

4.1 HIBI ja Nios II

Erilaiset kommunikaatioväyläratkaisut ovat järjestelmäsuunnittelussa avainasemassa, koska eri komponentit viestivät toisilleen väylien avulla. Eräs tällainen väylä on TTY:ssa suunniteltu HIBI-väylä (engl. Heterogeneous IP Block Interconnection) [27]. HIBIn version 2 suunnitteluperiaatteina oli luoda helposti uudelleenkäytettävä, topologiariippumaton, skaalautuva ja tehokas väyläratkaisu komponenttien yhdistämiseen [27].

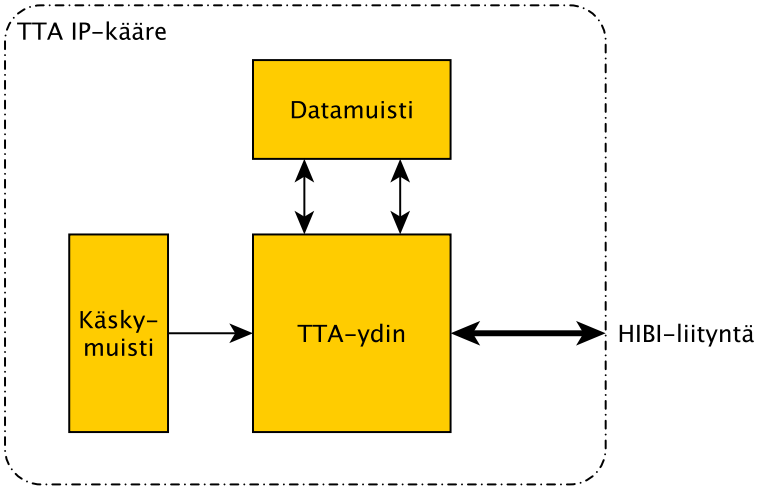
HIBI-väylä on myös keskeisessä osassa TTY:ssa Koski suunnitteluvuolla kehitetyissä järjestelmissä sekä vuota silmälläpitäen kehitetyissä IP-lohkoissa ja muissa komponenteissa. Esimerkiksi Nios II-prosessoria varten on tehty Nios to HIBI versio 2 (N2H2) IP-lohko, jonka avulla Nios II-prosessori voidaan kytkeä HIBI-väylään. N2H2-lohko tukee DMA-siirtoja (engl. Direct Memory Access) HIBI-väylän lävitse, jolloin prosessorin ei tarvitse aktiivisesti hoitaa datasiirtoja. N2H2-lohkon ja Nios II:n väliset liitynnät on esitetty kuvassa 4.1. Koska Nios II on Alteran tarjoama pehmytydinprosessori, jonka pääliityntä on Alteran Avalon Memory Mapped Master Interface [28] eli Avalon muistikartoitettu isäntäliityntä, on luonnollista, että N2H2



Kuva 4.1: Nios II:n ja N2H2-komponentin esimerkkikytkentä. Nios II on kytketty N2H2-komponenttin konfigurointiliityntään ja kaksiporttimuistin (DP RAM) ensimmäiseen porttiin. N2H2-komponentin lähetys- ja vastaanottoportit on kytketty arbitterin kautta muistikompentin toiseen porttiin.

on kytketty prosessoriin muistikartoitetun Avalon-väylän avulla. Täten N2H2-lohkon komentoliityntä on toteuttu Avalon muistikartoitetulla orjaliitynnällä. Kuten kuvasta 4.1 käy ilmi, kokonaisuuteen kuuluu myös niin sanottu kaksiporttinen *scratch pad memory* eli työmuisti, joka on oleellinen DMA-siirtojen kannalta. N2H2:n Avalon isäntäliitynnät mahdollistavat sen, että N2H2 pystyy lukemaan ja kirjoittamaan dataa tähän työmuistiin, jolloin se kykynee toteuttamaan DMA-siirtoja. Toista muistiporttia puolestaan käyttää Nios II. Kahden muistiportin avulla N2H2 ja Nios II pystyvät käyttämään muistia samaan aikaan, joka nostaa suorituskykyä esimerkiksi tilanteessa, jossa N2H2 hoitaa uutta DMA-siirtoa työmuistiin samalla, kun Nios II käsittelee edellisen siirron dataa.

Jotta N2H2-lohkoa voitaisiin käyttää Nios II:lle kirjoitetusta ohjelmakoodista, lohko tarvitsee ajurit. Muistikartoitetun liitynnän luonteeseen kuuluu se, että oheislaitetta ohjataan kirjoittamalla ja lukemalla komentorekistereitä, joille on annettu muistiosoite. Näin ollen myös N2H2-lohkoa ohjataan kirjoittamalla orjaliitynnän kautta lohkon kontrollirekistereihin. Matalan tason ajurikoodit on toteutettu yksinkertaisia operaatioita tekevien C-makrojen ja näitä makroja yhdistelevien C-funktioiden avulla. Näiden funktioiden avulla on mahdollista konfiguroida HIBI-



Kuva 4.2: Koskivuohon yhteensopivaksi käärityy TTA-IP-lohko.

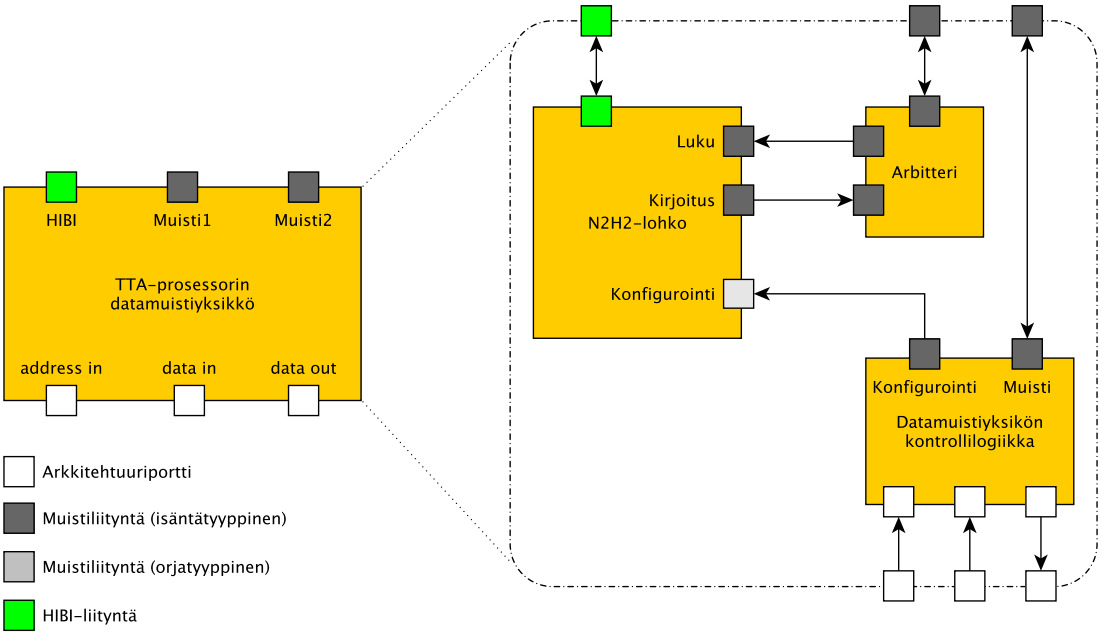
kanavia vastaanottamista varten, lähettää dataa DMA-siirtoina ja tarkastella, onko lohko vastaanottanut datalähetyksen. Matalan tason ajureiden lisäksi on olemassa eCos-reaaliaikakäyttöjärjestelmää varten toteuttu korkeamman tason ajurirajapinta, joka mahdollistaa keskeytyspohjaisen käsittelyn. Tämä mahdollistaa esimerkiksi sovelluskohtaisen vastaanottokeskeytyksenkäsittelyn, jolloin prosessorin ei tarvitse aktiivisesti tarkastella, onko datalähetyksiä saapunut.

4.2 TTA:n integroiminen Koskivuohon

Minimivaatimus TTA-prosessorin integroimiseksi Koskivuohon on laitteistotason yhteensopivuuden takaaminen, joka onnistuu toteuttamalla HIBI-liitynnän TTA:lle. Suunnittelutyön helpottamisen kannalta olisi myös haluttavaa, että TCE-työkalut osaisivat kääriä TTA-prosessorin valmiiksi IP-lohkoksi ja generoida tälle lohkolle IP-XACT-kuvauksen, jolloin TTA-prosessorin käyttöönotto Koskivuon työkaluissa vaatisi mahdollisimman vähän suunnittelijan tekemää käsityötä. Kuva 4.2 ilmaisee yksinkertaisuudessaan, miltä TTA-prosessorin tulisi näyttää Koskivuohon sopivaksi IP-lohkoksi käärimisen jälkeen.

4.2.1 Laitteistotason integroiminen

Laitteistotason yhteensopivuuden takaavan HIBI-liitynnän tekemisessä ei haluttu ruveta keksimään pyörää uudelleen, vaan pyrittiin uudelleenkäyttämään jo aiemmin toteutettua ja testattua N2H2-lohkoa. Tämä voisi mahdollistaa myös sovelluskoodin siirrettävyyden Nios II:n ja TTA:n välillä, koska sovellus voisi käyttää samoja ajurikoodeja N2H2:n käsittelyyn. Muistikartoituksen ja työmuistin tarpeen



Kuva 4.3: HIBI-yhteensopivan TTA datamuistiyksikkö ja sen sisäinen toteutus.

vuoksi tuntui luonnolliselta vaihtoehdolta sijoittaa N2H2-komponentti datamuistiyksikön yhteyteen. Tässä tapauksessa myös niin sanottu työmuisti voisi toimia samalla koko prosessorin datamuistina, jolloin voitaisiin välttää kokonaan työmuistin ja päädatamuistin väliset muistikopioinnit. Ulkoisesti tarkasteltuna tällaisessa HIBI-yhteensopivassa datamuistiyksikössä olisi kolme erillistä ulkoista liityntää, yksi HIBI-liityntä ja kaksi muistiliityntää, kuten kuvan 4.3 vasen puolisko havainnollistaa. Näistä HIBI-liityntä ja yksi muistiliityntä ovat peräisin H2N2-lohkolta ja toinen muistiliityntä on TTA:n tavanomaista datamuistikäyttöä varten.

N2H2-lohkoa ei voi kuitenkaan ottaa aivan suoraan käyttöön Avalon-liityntöjen vuoksi. Nios II:sta käytettäessä Alteran SOPC Builder-työkalu hoitaa Avalon-liityntöjen kytkemisen toisiinsa ja tarvittaessa arbittereiden lisäämisen kytkentöjen väliin. TTA:n tapauksessa haluttiin välttää kolmannen osapuolen työkalun käyttämistä, koska sen generoimissa tiedostoissa oli vapaan levittämisen estäviä lisenssiehtoja. Toisaalta ei ole myöskään haluttava vaihtoehto jättää Avalon-väylän generointia suunnittelijan harteille, koska se estäisi vuon automatisoinnin. Ongelman pystyi kuitenkin kiertämään tekemällä käsin Avalonin kaltaisen väyläratkaisun. Kuten kuvasta 4.1 käy ilmi, tarvittavia kytkentöjä on vähän, joten väylän rakenne on yksinkertainen. Lisäksi Avalonin muistikartoitetun luonteen vuoksi liitynnät on helppo sovittaa TTA:n ominaiseen muistiliityntämalliin. Kuvan 4.3 oikea puolisko havainnollistaa datamuistiyksikön sisäisiä kytkentöjä. Kytkennät voitiin tehdä suoraan N2H2-lohkon luku- ja kirjoitusliityntää lukuunottamatta, sillä niiden pitää

käyttää samaa muistiliityntää. Tämän vuoksi ne täytyi kytkeä arbitteriin, joka sarjallistaa samanaikaiset operaatiot antamalla suoritusvuoron aina toiselle liitynnälle, suosien lukuoperaatioita. Datamuistiyksikön kontrollilogiikkaa piti muuttaa siten, että luku- ja kirjoitusoperaatiot saadaan jaettua oikein datamuistin ja N2H2-lohkon kontrollirekisterien kesken. Muistikartoitus tehtiin siten, että 32-bittisen osoitteen ylin bitti ilmaisee, kumpaan komponenttiin operaatio kohdistuu. Ylimmän bitin arvolla nolla, operaatio käsittelee datamuistia ja arvolla yksi operaatio ohjautuu N2H2-lohkolle. Näiden muokkauksien jälkeen myös N2H2:n matalan tason ajurit voidaan ottaa käyttöön ilman niihin kohdistuvia muutoksia, sillä N2H2-lohkon kontrollirekisterien alkuosoite on parametrisoitu ajurikoodissa. Tässä tapauksessa kontrollirekisterien alkuosoite on heksadesimaalina 0x80000000 eli 32-bittisen osoitteen ylin bitti on 1.

Kaikenkaikkiaan N2H2-lohkon integroiminen TTA:n datamuistiyksikköön onnistui helposti. HIBI-yhteensopivan datamuistiyksikön toteuttamiseen tarvittiin noin 857 riviä kommentoitua VHDL-koodia, josta reilu 300 riviä on uudelleenkäytettyä koodia TTA:n alkuperäisestä datamuistiyksiköstä. Koodiriveihin ei ole laskettu tiedostojen alussa olevia lisenssiehtoja eikä N2H2-lohkon vaatimia koodirivejä. Vertailun vuoksi N2H2-lohkon toteutuksessa on noin 1301 riviä VHDL-koodia, joten olemassa olevan N2H2-lohkon uudelleenkäyttämisellä säästettiin huomattava määrä suunnittelu- ja varmennustyötä.

4.2.2 Koski Integrator

Suunnittelun helpottamiseksi ja integroinnin automatisoinniksi kehitettiin myös Platform Integrator-komponentti nimeltään Koski Integrator. Tämän komponentin tehtävänä on liittää HIBI-yhteensopivaan TTA-prosessoriytimeen käsky- ja datamuisti ja luoda prosessorin ja muistien ympärille niin sanottu IP-kääre. Ulkopuolelta tarkasteltuna tämä kääritty TTA-prosessori näyttää tällöin mustalta laatikolta, jonka rajapinnasta löytyy HIBI-liityntä, kuten kuvasta 4.2 käy ilmi. Yhteensopivuuden takaamiseksi Koski-työkaluihin Koski Integrator luo TTA IP-kääreelle automaattisesti IP-XACT-kuvauksen.

Koski Integrator toteutettiin C++-kielellä käyttäen hyväksi TCE-työkalujen Platform Integrator-rajapintaa ja -luokkia. Integraattorin toteuttamiseen tarvittiin 1212 koodiriviä, joista IP-XACT:n mallintamiseen ja käsittelemiseen kului 1005 riviä. Integraattorin lisäksi piti toteuttaa myös kaksiporttisen RAM-muistin generoiva luokka, joka onnistui 280 koodirivillä. Kaikenkaikkiaan toteutukseen kirjoitettiin 1492 koodiriviä. Riveihin on laskettu mukaan myös kommentit, muttei kooditiedostojen alussa olevia lisenssiehtorivejä.

5. SUUNNITTELUESIMERKKI

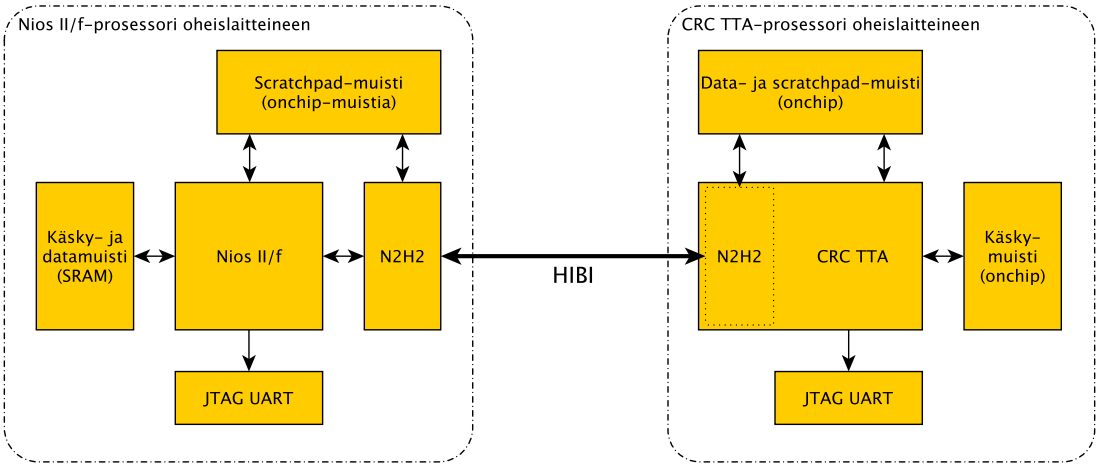
Tässä kappaleessa käydään läpi HIBI-yhteensopivan TTA-prosessorin suunnitteluvaiheet suunnitteluesimerkin avulla. TTA-prosessori liitetään HIBI-väylän kautta järjestelmään, jossa on Nios II/f-prosessori. Nios II-prosessorilla eCos-reaaliaikakäyttöjärjestelmää. Tämä järjestelmä on kuvattu lohkokaaviokuvassa 5.1. Järjestelmän kellotaajuus on asetettu 50 MHz:iin ja yksinkertaistamisen vuoksi tämä valitaan myös TTA:n kellotaajuudeksi. Järjestelmä tullaan syntesoimaan Stratix II DSP Pro FPGA-laudalle.

5.1 Testisovellus CRC-32

TTA:lla suoritettava testisovellus laskee annetulle datalle 32-bittisen CRC:n (Cyclic Redundancy Check) eli tarkistustiivisteen, jonka avulla voidaan todeta datan eheys vaikkapa datasiirron jälkeen. Esimerkiksi ethernet-kehys sisältää 32-bittisen CRC:n, jonka avulla vastaanottaja tarkistaa, että saapunut ethernet-paketti vastaa lähetettyä pakettia [29]. TTA-prosessori tulee suunnitella sellaiseksi, että se kykenee laskemaan CRC-tiivisteen ethernet-paketin minimi- ja maksimikoon välillä vaihtelevalle datamäärälle. Datamäärästä voidaan poistaa 32-bittinen CRC-kenttä, koska TTA:n on tarkoitus laskea se, joten datan koko vaihtelee 60 ja 1518 tavun välillä [29].

Sovelluskoodina käytettiin valmista, avoimena lähdekoodina julkaistua CRC-toteutusta, joka on Michael Barrin tekemä [30]. Koodi sisältää kolme eri CRC versiota sekä hitaan ja nopean tavan laskea CRC-tiiviste. Tässä esimerkissä keskitytään vain CRC-32 versioon, jota käytetään myös ethernetin tapauksessa. Lisäksi esimerkissä keskitytään vain nopeaan laskentatapaan, joka hyödyntää taulukkoon tallennettuja välituloksia laskennan nopeuttamiseksi. Samaa sovellusta käytetään myös TCE-työkalujen esittelytutoriaalissa [31].

Suunnittelun ensimmäinen vaihe on varmennusvuokuvan 3.6 mukaisesti tuottaa referenssituloksia työpöytäympäristössä. Näin ollen sovelluskoodia mu§okattiin sellaiseksi, että se generoi satunnaista laskentadataa taulukkoon ja laskee datalle CRC-tiivisteen. Tämän lisäksi sovellus tallentaa laskentadatan C-taulukon muodossa otsikkotiedostoon, jolloin TTA:lla voidaan käyttää samaa laskentadataa. Lisäksi otsikkotiedostoon kirjoitetaan myös oikea laskentatulos, jotta tuloksien verifiointi voidaan automatisoida. Staattisen laskentadatan käyttämisessä on kuitenkin mahdollista vajota sudenkuoppaan, sillä nokkela kääntäjä saattaa optimoida koodin sisääntulevan



Kuva 5.1: Esimerkkijärjestelmän lohkokaavio

datan perusteella, vaikka oikeasti data voi olla mitä vain. Tämä voidaan kuitenkin välttää käyttämällä testidatassa C:n avainsanaa *volatile*, joka pakottaa kääntäjän tekemään sellaista koodia, joka aina lukee muuttujan arvon muistista sen sijaan, että se käyttäisi vakioarvoa.

Seuraava vaihe oli siirtää sovelluskoodi TTA:lle. Tässä välissä sovelluskoodiin tehtiin vain hienoisia muutoksia. Laskentadatan generointi poistettiin ja sen tilalle tuli otsikkotiedoston sisällyttäminen. Pääohjelman loppuun lisättiin myös laskentatuloksen varmennus, joka vertaa laskettua CRC-tiivistettä otsikkotiedostosta löytyvään oikeaan tiivisteeseen. Vertailun tuloksesta riippuen tulostetaan joko 'O', jos tulos oli oikein, tai 'N', jos tulokset erosivat toisistaan. Tämän jälkeen tulostetaan vielä itse CRC-tiiviste heksadesimaalina. Molemman tulosteet on myös mahdollista kytkeä pois päältä C:n esiprosessorikäskyjen avulla, jolloin tulosteita ei jää lopulliseen sovellukseen. Tämä mahdollistaa tarkemman kellojaksojen määrän keräämisen, sillä tulostaminen luonnollisesti kasvattaa kellojaksolukemaa. Tuloksen verifiointi on arkkitehtuurisimulaatiossa mahdollista myös ilman tulosteita, sillä simulaattorissa voidaan katsoa suoraan tulosmuuttujan sisällöstä, onko se oikein vai ei. Tässä vaiheessa suunnittelua ei vielä keskitytä HIBI-kommunikaation tekemiseen, vaan tavoitteena on muokata arkkitehtuuri mahdollisimman sopivaksi itse sovelluksen suorittamiseen.

5.2 TTA-prosessorin suunnittelu

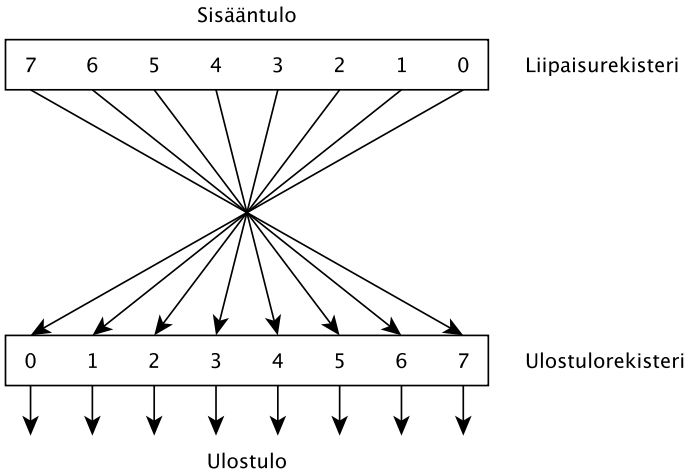
Suunnitteluvuokuvan 3.1 mukainen iterointi aloitetaan niin sanotulla minimaalisella arkkitehtuurilla, joka on kaikkein pienin arkkitehtuuri, jolle tcecc-kääntäjä pystyy kääntämään standardin mukaista C-koodia. Tämän arkkitehtuurin sisältämät resurssit on lueteltu taulukossa 5.1. Arkkitehtuurin suorituskyky on vaatimaton, mut-

Taulukko 5.1: Minimaalisen TTA-prosessorin resurssit

Resurssi	**kpl**	**Selite**
ALU	1	Laskentayksikkö, jossa operaatiot: add, sub, eq, gt, gtu, and, ior, xor, shl, shr, shru, sxhw, sxqw
IO	1	Laskentayksikkö, joka sisältää stdout-operaation
Datamuistiyksikkö	1	Sisältää operaatiot: ldw, ldh, ldhu, ldq, ldqu, stw, sth, stq
Rekisteripankki	1	Sisältää 5 kappaletta 32-bittistä rekisteriä
Boolean rekisteripankki	1	Sisältää 2 kappaletta 1-bittistä rekisteriä
Kontrolliyksikkö	1	Ohjaa prosessorin suoritusta ja toteuttaa hyppy- ja kutsukäskyn
Siirtoväylä	1	Operandien siirtoväylä

ta minimaalinen arkkitehtuuri kuitenkin tarjoaa valmiin lähtökohdan, jolloin arkkitehtuuria ei tarvitse alkaa tekemään tyhjästä. Sovelluskoodissa valitaan laskettavaksi CRC-tiiviste 1 518 tavun kokoiselle datamäärälle, joka on tavallisen ethernetkehyksen maksimikoko ilman CRC-tiivistettä [29]. Kääntämisen ja simulaation jälkeen saadaan sovelluksen suoritusaika, joka on 436 822 kellojaksoa, sisältäen automaattisen verifikaation. Prosessorin resurssienkäyttöasteesta on huomattavissa, että suorituksen aikana tehdään todella paljon lataus- ja tallennusoperaatioita datamuistiin. Tämä kielii siitä, että arkkitehtuurissa on liian vähän rekistereitä, jolloin koodissa pitää väliaikaisesti tallentaa muuttujia pinoon. Kasvattamalla rekisteripankin rekisterimäärää viidestä kuuteentoista, kellojaksolukema tippuu 138 637 kellojaksoon. Nykyisellään arkkitehtuuri ei kuitenkaan voi juuri hyödyntää rinnakkaisuutta, koska siirtoväyliä on vain yksi. Voidaan nopeasti huomata, että kahdella siirtoväylällä suoritettujen kellojaksojen määrä tippuu lukemaan 82 312 ja kolmella siirtoväylällä suoritus kestää enää 56 452 kellojaksoa, joka on enää noin 13% alkuperäisestä kellojaksolukemasta. Samalla logiikkaelementtien (LE) käyttö kasvoi noin 43%, kuten synteesitulostaulukosta 5.4 käy ilmi.

Kun arkkitehtuurin ilmeisimmät pullonkaulat on korjattu, on hyvä analysoida sovellusta hieman tarkemmin ja profiloida suoritusta. Analyysin tuloksena käy ilmi, että suurin osa suoritusajasta, noin 77%, tuhlaantuu `reflect()`-funktiossa, joka nimensä mukasesti peilaa sille annetun bittikuvion. Tarkempi tutkiskelu paljastaa, että funktiota kutsutaan jokaiselle laskettavalle tavulle ja laskennan lopuksi vielä koko 32-bittinen tiiviste peilataan ennen viimeistä xor-operaatiota. Syy, miksi tämä peilausoperaatio on hidas, johtuu siitä, että operaatio tehdään koodissa iteratiivisesti yksi bitti kerrallaan. Sen sijaan laitteistolla tehtynä tämä operaatio on hyvin yksinkertainen. Se voidaan toteuttaa yksinkertaisesti kytkemällä johtimia ris-



Kuva 5.2: Esimerkki operaation *REFLECT8* eli 8-bittisen bittikuvion peilaamisen laitteistototeutuksesta. Numerot kuvaavat bittikuvion bitti-indeksejä.

tiin ja rekisteröimällä ulostulo. Kuva 5.2 havainnollistaa peilausoperaation tekemistä 8-bittiselle bittikuviolle. Peilausoperaatiot vaikuttavat varteenotettavilta erikoisoperaatioilta, joten seuraava askel on erikoisoperaatiovuon (kuva 3.4) mukaisesti kääntäjämäärittelyn ja simulaatiomallien tekeminen. Pitää siis luoda kaksi operaatiota: *REFLECT8*, joka on peilaus 8-bittiselle bittikuviolle, ja *REFLECT32*, joka peilaa 32-bittisen bittikuvion. Operaatiolla on täten 1 sisäänmeno ja 1 ulostulo. Simulaatiomallissa voi puolestaan hyödyntää alkuperäistä reflect-funktiota. Sovelluskoodin muokkaaminen on myös yksinkertaista, sillä reflect-funktiokutsut korvataan erikoisoperaatiomakroilla `_TCE_REFLECT8()` ja `_TCE_REFLECT32()`. Viimeinen vaihe ennen kääntämistä on operaatioiden lisääminen arkkitehtuuriin. Molemmat operaatioit voidaan tehdä samalla laitteistolla, kun peilauksen leveys mitoitetaan leveämmän operaation mukaan 32-bittiseksi. Kahdeksanbittisen luvun tapauksessa peilauksen tulos pitää siirtää ylimmiltä biteiltä takaisin alimmille biteille. Koska molemmat operaatiot voidaan toteuttaa samalla laitteistolla, operaatiot voidaan sijoittaa samaan laskentayksikköön. Lisäksi peilausoperaatiot ovat niin yksinkertaisia, että operaatiot on mahdollista tehdä yhdessä kellojaksossa. Täten operaatioiden latenssiksi asetetaan yksi kellojakso. Nyt simulaatiotuloksista huomataan, että erikoisoperaatioiden avulla suoritus kestää enää 21 273 kellojaksoa. Koska parannus on merkittävän suuri ja näiden erikoisoperaatioiden toteuttaminen laitteistonkuvauskielellä on triviaalia, erikoisoperaation toteuttaminen on kannattavaa.

Prosessorin resurssienkäytön jatkotarkastelu paljastaa, että ALU-laskentayksikön yhteenlaskua, yhtäsuuruusvertailua, shiftausoperaatioita sekä loogisia and- ja xor-operaatioita käytetään erityisen paljon. On hyvin todennäköistä, että osa näistä

Taulukko 5.2: Monoliittisen ALU:n korvaavat laskentayksiköt

Yksikkö	Operaatiot
alu_comp	add, sub, eq, gt, gtu
logic	and, ior, xor
shifter	shl, shr, shru

operaatioista voitaisiin laskea rinnakkain, joten on järkevää kokeilla palastella monoliittinen ALU useaksi eri laskentayksiköksi. Rautatietokannasta löytyykin ALU:n korvaamiseen sopivat laskentayksiköt, jotka on lueteltu taulukossa 5.2. Huomion arvoista on, että puolisanan ja neljännessanan etumerkkilaajennusoperaatiot (sxhw ja sxqw) voitiin jättää pois, koska niitä ei käytetty suorituksen aikana laisinkaan. Laskentayksiköiden hajauttamisen jälkeen suoritetut kellojaksot pienenivätkin lukemaan 19 755.

Tässä vaiheessa alkaa vaikuttamaan siltä, että suorituskyky ei enää parane, vaikka prosessorin resursseja kasvatettaisiin. Esimerkiksi laskentaresurssien, rekisteripankkien ja siirtoväylien kaksinkertaistaminen ei saa muutosta aikaan kellojaksolukemassa. Tämä johtuu siitä, että sovelluskoodista ei löydy enempää käskytason rinnakkaisuutta, jota lisäresurssit voisivat hyödyntää. Toisaalta aikaisessa vaiheessa vastaantuleva raja käskytason rinnakkaisuudelle ei ole kovin yllättävä, sillä sovelluskoodin koko on pieni. TTA:lle käännetyssä ohjelmassa on vain 41 käskyä, jos tuloksen verifiointi on käytössä, ja 35 käskyä, jos tuloksen verifiointi on pois päältä. Arkkitehtuurin resurssit ennen HIBI-kommunikaation toteuttamista ja lopullista viilausta on lueteltu taulukossa 5.3. Taulukossa 5.4 on lueteltu tämän arkkitehtuurin kustannustietoja (rivi CRC TTA 1) Stratix II FPGA:lle syntesoituna käyttäen tavanomaista datamuistiyksikköä HIBI-yhteensopivan datamuistiyksikön sijaan. Tämä sen vuoksi, että suunnitteluvuon loppuvaiheessa käytettiin Stratix II DSP Pro FPGA-laudalle suunnattua Platform Integrator-komponenttia, jonka avulla prosessorin ja sovelluksen itsenäinen toimivuus voitiin verifioida FPGA:lla. Vertailun vuoksi taulukon 5.4 ensimmäisellä rivillä on minimalistisen arkkitehtuurin synteesitulokset.

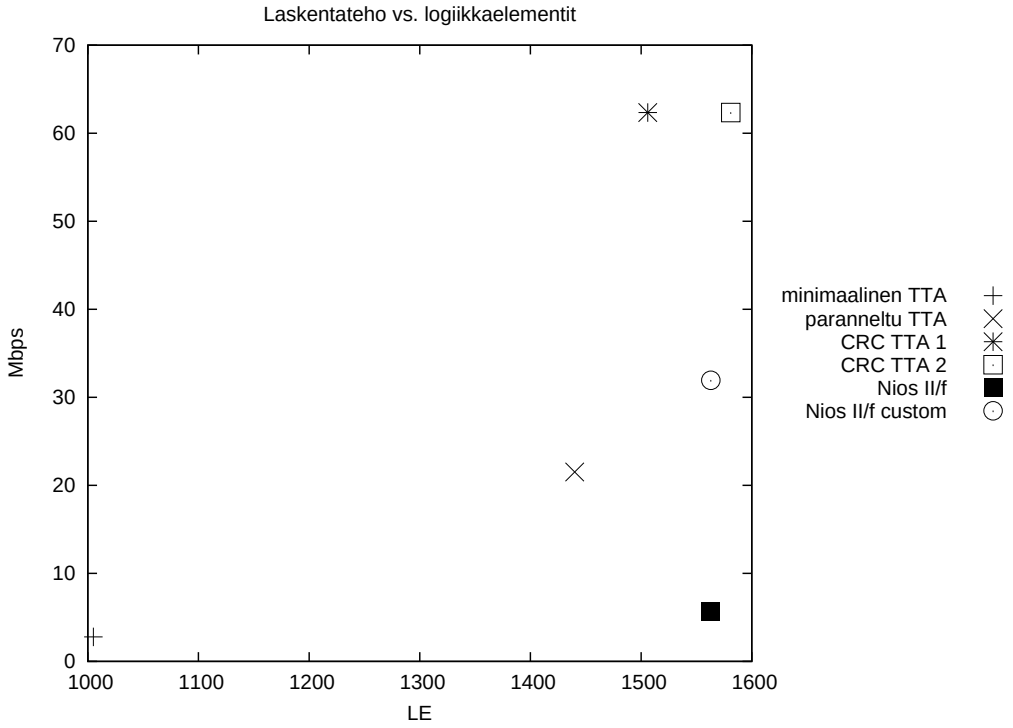
5.3 Resurssienkäytön optimointi

Taulukon 5.5 riviltä CRC TTA 1 käy kuitenkin ilmi, että prosessorin käskynleveys on varsin suuri, 126 bittiä. Suuri leveys johtuu siitä, että jokaisella siirtoväylällä on mahdollista siirtää 32-bittisiä vakioarvoja. Vakioarvon siirtäminen vaatii väylän käskyformaatin lähdekentästä vähintään vakioarvon leveyden verran bittejä, joten tässä tapauksessa jokaisen väylän lähdekentän on vähintään 32 bittiä leveä. Käskynleveyttä voidaan lähteä pienentämään käyttämällä vakioarvoyksikköä, jolloin väyliltä voi-

Taulukko 5.3: CRC:n laskemiseen optimoidun TTA-prosessorin resurssit

Resurssi	**kpl**	**Selite**
Alu_comp	1	Laskentayksikkö, jossa operaatiot: add, sub, eq, gt, gtu
Logic	1	Laskentayksikkö, jossa operaatiot: and, ior, xor
Shifter	1	Laskentayksikkö, jossa operaatiot: shl, shr, shru
Reflecter	1	Laskentayksikkö, jossa erikoisoperaatiot: reflect8 ja reflect32
IO	1	Laskentayksikkö, joka sisältää stdout-operaation
Datamuistiyksikkö	1	Sisältää operaatiot: ldw, ldh, ldhu, ldq, ldqu, stw, sth, stq
Rekisteripankki	1	Sisältää 16 kappaletta 32-bittistä rekisteriä
Boolean rekisteripankki	1	Sisältää 2 kappaletta 1-bittistä rekisteriä
Kontrolliyksikkö	1	Ohjaa prosessorin suoritusta ja toteuttaa hyppy- ja kutsukäskyn
Siirtoväylä	3	Operandien siirtoväylä

daan pienentää tai poistaa kokonaan vakioarvosiirrot. Kun siirtoväylien vakioarvosiirrot korvataan vakioarvoyksiköllä, käskynleveys pienenee 43 bittiin. Muutoksella on kuitenkin negatiivinen vaikutus suorituskykyyn, koska jokainen vakioarvo pitää kirjoittaa omalla käskyllä vakioarvoyksikköön ja siirtää yksiköstä kohteeseen. Suoritukseen kuluikin nyt 54 690 kellojaksoa. Kääntäjän tuottamaa koodia tarkastelemalla voidaan huomata, että suurin osa käytetyistä vakioarvoista on pieniä, positiivia tai negatiivisia lukuja. Lisäksi suurimman osan näistä luvuista voi esittää 13-bittisen etumerkillisen luvun avulla eli arvoalueella [-4096, 4095]. Lisäämällä ensimmäiseen siirtoväylään tuen 13-bittisille etumerkillisille vakioarvoille kasvattaa käskynleveyden 52 bittiin. Lisäksi toiseen siirtoväylään lisättiin tuki 5-bittisille etumerkittömille vakioarvoille eli arvoalueelle [0,31] ja kolmanteen väylään tuki 2-bittisille etumerkillisille luvuille eli arvoalueelle [-2,1]. Tällöin käskynleveydeksi muodostui 53 bittiä. Arkkitehtuurisimulaatio paljastaa, että näillä muutoksilla suorituskyky palautuu takaisin 19 755 kellojaksoon. Käännetyn koodin lisätutkiminen paljastaa vielä, että ehdollisia siirtoja tarvitaan korkeintaan kahdella väylällä samaan aikaan. Täten kolmannelta väylältä voidaan poistaa tuki ehdollisille siirroille, jolloin käskynleveydestä säästetään 3 bittiä. Arkkitehtuurisimulaatio varmentaa, ettei muutoksella ollut vaikutusta suorituskykyyn. Lopuksi tämä arkkitehtuuri vielä syntesoitiin ja verifioitiin suoritus FPGA:lla. Synteesitulokset on ilmoitettu taulukon 5.4 rivillä CRC TTA 2. Vakioarvoyksikön lisääminen arkkitehtuuriin kasvattaa LUT:ien ja rekistereiden käyttöä, mutta kokonaisuudessaan rekisterikulutus pieneni, koska siir-



Kuva 5.3: Eri arkkitehtuurien suorituskyky CRC:n laskemisessa verrattuna logiikkaelementtien (LE) käyttöön. Suorituskyky on mitattu, kuinka monelle megabitille dataa prosessori pystyy laskemaan CRC-tiiviseen sekunnissa (engl megabits per second, Mbps).

toväylien vakioarvokentät pienenivät. Taulukosta 5.4 on myös huomattavissa, että maksimikellotaajuus laski tällä arkkitehtuurilla. Osaltaan tämä johtuu synteesityökalun ominaisuuksista. Jos synteesille on määritelty jokin kohdekellotaajuus, synteesityökalu pyrkii pääsemään vähintään tähän kellotaajuuteen. Kun tämä kohdekellotaajuus on ylitetty, synteesityökalu ei oletusarvoisesti enää keskity nopeutta parantaviin optimointeihin, jolloin saavutettu kellotaajuus ei välttämättä ole suurin saavutettavissa oleva kellotaajuus, jolla järjestelmä voisi toimia. Yksittäisten TTA-prosessorien syntesoinneissa tämä kohdetaajuus oli 100 MHz, joka on FPGA-laudan kellokiteen vakiotaajuus.

5.4 Suorituskykyanalyysi

TTA-prosessorin muokkaamisella saatiin pienennettyä 1 518 tavun datasta CRC-tiivisteen laskemiseen kuluva kellojaksolukema 19 748, joka on noin 4,5% minimaalisen TTA:n tarvitsemasta kellojaksolukemasta, kuten taulukosta 5.5 käy ilmi. Samalla logiikkaelementtien käyttö kasvoi lukemaan 1 581 eli vain 57%.

Vertailunkohdan saamiseksi CRC-tiivisteen laskeminen toteutettiin myös Nios II/f-prosessorilla käyttäen samaa sovelluskoodia ja samaa laskentadataa. Nios II-prosessorilla kesti 215 872 kellojaksoa laskea CRC-tiiviste 1 518 tavulla dataa 100 MHz kel-

lotaajuudella. Lisäksi toteutin samat *REFLECT*-erikoisoperaatiot Nios II-prosessorin käskykantalaajennuksen avulla, jolloin suoritusaika tippui 38 037 kellojaksoon. Lukema on silti noin 48% suurempi kuin CRC TTA 2:lla. Lisäksi Nios II/f ja CRC TTA 2 ovat hyvinkin saman kokoisia logiikkaelementeillä mitattuna, sillä Nios II/f tarvitsi 1 563 LE:iä, joka on 18 LE:tä vähemmän kuin CRC TTA 2:n koko.

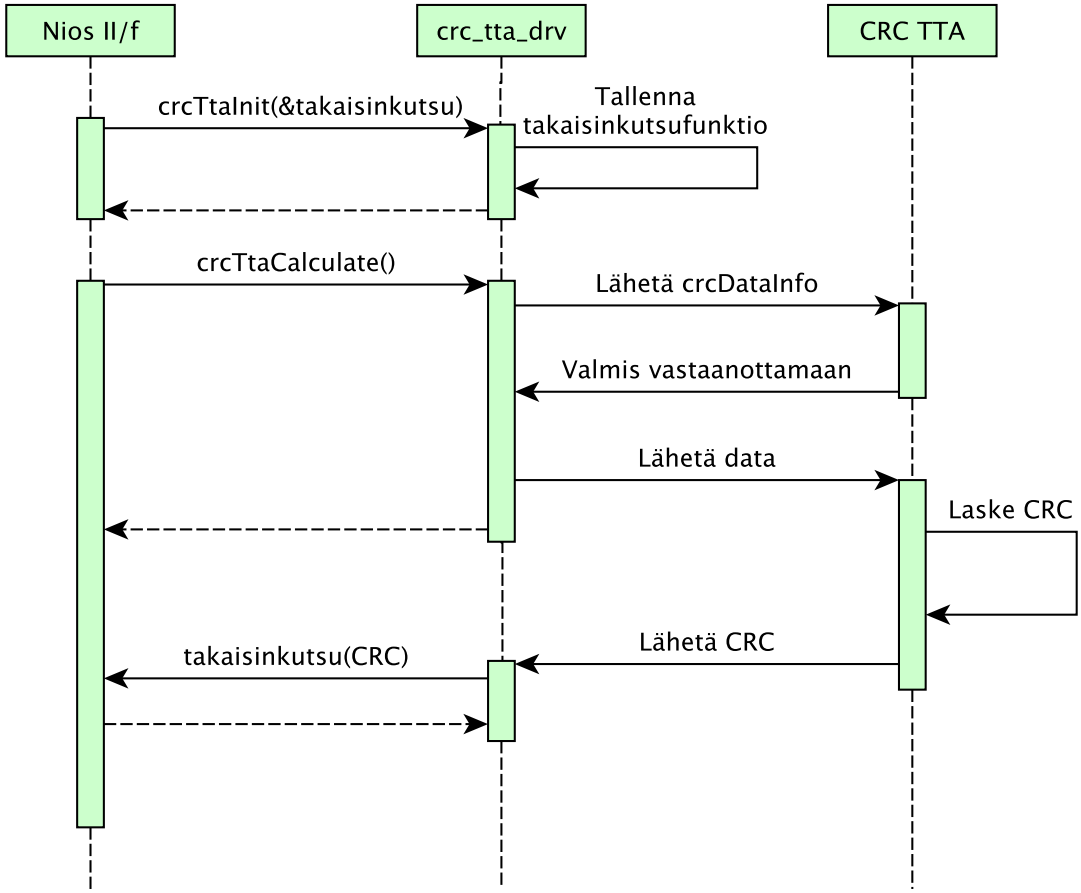
Arkkitehtuurien eron havainnollistamiseksi piirrettiin kuvaaja 5.3, josta selviää CRC-tiivisteen laskentatehon suhde käytettyihin logiikkaelementteihin. Laskentatehon mittarina käytettiin sitä, kuinka monelle megabitille dataa (engl. megabits per second, Mbps) prosessori kykenee laskemaan CRC-tiivisteen sekunnissa. Lukema on mitattu prosessoimalla 1 518 tavua eli 12 144 bittiä dataa per CRC-tiiviste 100 MHz:n kellotaajuudella.

5.5 HIBI-kommunikaation suunnittelu ja toteutus

Jotta TTA-prosessoria voisi hyödyntää komponenttina järjestelmässä, pitää suunnitella ja toteuttaa sovellustason kommunikaatioprotokolla, joka käyttää HIBI:ä. Koska datamäärä voi vaihdella, Nios II-prosessorin pitää ensin ilmoittaa siirrettävän datan koko, jotta TTA voi alustaa oikeanlaisen HIBI-vastaanottokanavan. Samoin myös CRC-tiivisteen laskemisessa tarvitaan tieto, kuinka monta tavua datamäärään kuuluu. Tässä vaiheessa pitää kuitenkin huomioida HIBI-siirtoihin liittyvät ominaisuudet, nimittäin HIBI lähettää 32-bittisiä sanoja kerrallaan. Laskentadata voi puolestaan loppua mihin tahansa tavurajaan, joten siirrettävään dataan on tarvittaessa lisättävä tyhjää dataa, jotta sanaraja täyttyisi. Tämän TTA:lle pitää lähetetään sekä laskentadata koko tavuina että siirrettävän datan määrä 32-bittisinä sanoina seuraavanlaisen tietueen avulla:

```
struct crcDataInfo {
   uint32_t sizeInBytes;
   uint32_t sizeInWords;
};
```

Kun ohjelman suoritus alkaa, TTA-prosessori valmistautuu vastaanottamaan tällaisen *crcDataInfo*-tietueen varaamalla sille HIBI-vastaanottokanavan. Kun Nios haluaa laskea CRC:n, se lähettää tietueen TTA:lle. Tämän jälkeen TTA analysoi saamansa tietueen, alustaa vastaanottokanavan dataa varten ja lopuksi viestii vastaanottovalmiudesta lähettämällä kuittauksen takaisin Nios:lle. Kuittauksen saatuaan Nios aloittaa datasiirron. Kun TTA on vastaanottanut data, se kutsuu CRC-tiivisteen laskemisfunktiota ja lähettää lasketun CRC-tiivisteen Nios:lle ja alustaa itsensä vastaanottamaan uuden *crcDataInfo*-tietueen.



Kuva 5.4: CRC:n laskemisen suoritusvaiheet ja kommunikaatio Nios II:n ja TTA:n välillä.

Jotta Nios II-prosessorin sovelluskoodista olisi helppo laskea CRC-tiiviste TTA-prosessorin avulla, kirjoitettiin *crc_tta_drv*-ajurimoduuli, joka hoitaa HIBI-kommunikoinnin Nios II:n ja TTA:n välillä. Ajurimoduuli siis abstrahoi varsinaisen kommunikaation ja tarjoaa sovelluskehittäjälle yksinkertaisen rajapinnan CRC:n laskemiseen. Kuvassa 5.4 on esitetty sekvenssikaavion avulla CRC:n laskemisen suoritusvaiheet ajurimoduulin avulla.

Nios II-sovelluksen alussa pitää alustaa *crc_tta_drv*-ajurimoduuli kutsumalla funktiota:

```
void
crcTtaInit(
       void (*callbackFunction)(uint32_t*, uint32_t, uint32_t));
```

Alustusfunktio ottaa parametrikseen funktio-osoittimen niin sanottuun takaisinkutsufunktioon. Ajurimoduuli saa eCos-käyttöjärjestelmältä keskeytyksen, kun se vastaanottaa TTA-prosessorilta CRC-tiivisteen. Tämän jälkeen ajurimoduuli palaut-

taa vastaanottamansa CRC-tiivisteen Nios II:n pääohjelmalle tämän takaisinkutsufunktion avulla. Alustusfunktiota tarvitsee kutsua vain kerran ohjelman suorituksen aikana.

Varsinainen CRC-tiivisteen laskeminen aloitetaan kutsumalla ajurifunktiota:

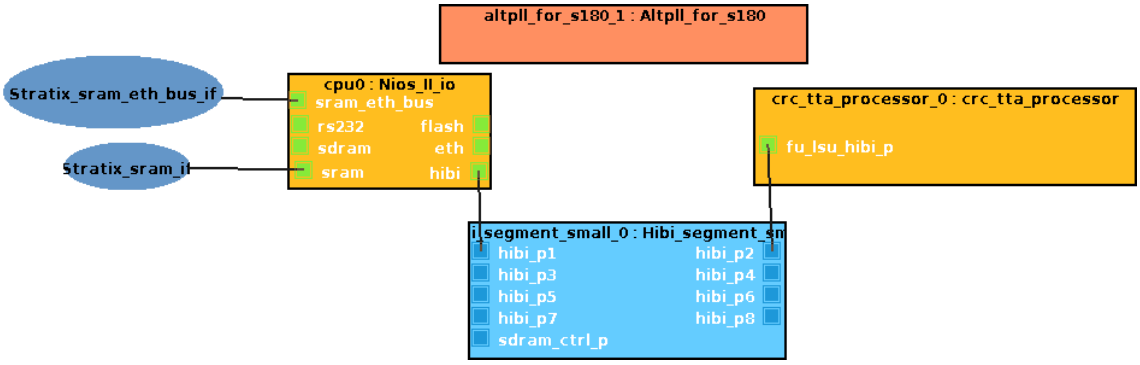
```
void
crcTtaCalculate(
       uint8_t* data,
       uint32_t sizeInBytes,
       uint32_t hibiAddr);
```

Funktio saa parametrina osoittimen laskentadataan, datan määrän tavuina sekä TTA-prosessorin HIBI-osoitteen. Ohjelman suoritus palaa tästä funktiosta takaisin kutsujalle siinä vaiheessa, kun datan DMA-siirto TTA:lle on saatu käynnistettyä. Tämä mahdollistaa sen, että Nios II voi suorittaa muuta koodia sillä aikaa, kun TTA-prosessori laskee CRC:tä. Vastaanottokeskeytyksen saapuessa ajurimoduuli kutsuu takaisinkutsufunktiota, jolloin CRC-tiiviste välitetään pääohjelmalle.

Vastaavasi myös TTA:n ohjelmakoodia piti muokata HIBI-kommunikaation mukaiseksi. Pääohjelmaan lisättiin funktio, joka hoitaa ajurimoduulia vastaavan protokollan mukaisen datan vastaanottamisen. CRC:n laskemisen jälkeen lisättiin koodia, joka hoitaa CRC-tiivisteen lähettämisen takaisin Nios II:lle. Muokkausten seurauksena TTA-ohjelman koko kasvoi 35 käskystä 188 käskyyn. Prosessoriarkkitehtuuriin ei tarvinnut tehdä muutoksia, ainoastaan datamuistiyksikön toteutukseksi piti valita HIBI-yhteensopiva datamuistiyksikkö. TTA:n käsky- ja datamuistiavaruuksien koot kuitenkin pienennettiin lopullisiin lukemiinsa. Käskymuistin osoiteleveydeksi piti valita 8 bittiä, jolloin muistiin mahtuu maksimissaan 256 käskyä. Osa muistista jää kuitenkin tässä tapauksessa käyttämättä, koska käskyjä on vähemmän. Ohjelma tarvitsee staattista datamuistia 2 905 tavun verran, jolloin pienin mahdollinen datamuistin koko on 4 096 tavua. Simulaation perusteella tämä riittää myös ajanaikaiselle muistikulutukselle eli käytännössä pinon tarpeisiin, koska ohjelmakoodissa ei käytetä dynaamista muistinvarausta. Lopullisen TTA-arkkitehtuurin synteesitulokset on listattu taulukon 5.4 rivillä CRC TTA 3, josta käy ilmi, että N2H2-komponentin lisääminen lähes kaksinkertaistaa LUT:ien ja rekisterien tarpeen.

5.6 Järjestelmän testaaminen

Seuraava askel on koota järjestelmä Koskivuon Kactus-työkalun avulla. Tätä varten TTA-prosessorista pitää tehdä HIBI-yhteensopiva komponentti, jolla on IP-XACT-kuvaus. Tämä onnistuu yksinkertaisesti generoimalla prosessorin uudelleen käyttämällä Koski Integrator:ia, joka hoitaa asian automaattisesti. Tämän jälkeen Koski Integrator:n tekemä TTA IP-lohko ja sitä vastaavan ajurimoduulin ohjelmakoodi



Kuva 5.5: Kactus-työkalun suunnitteluikkunan väylänäkymä testijärjestelmästä

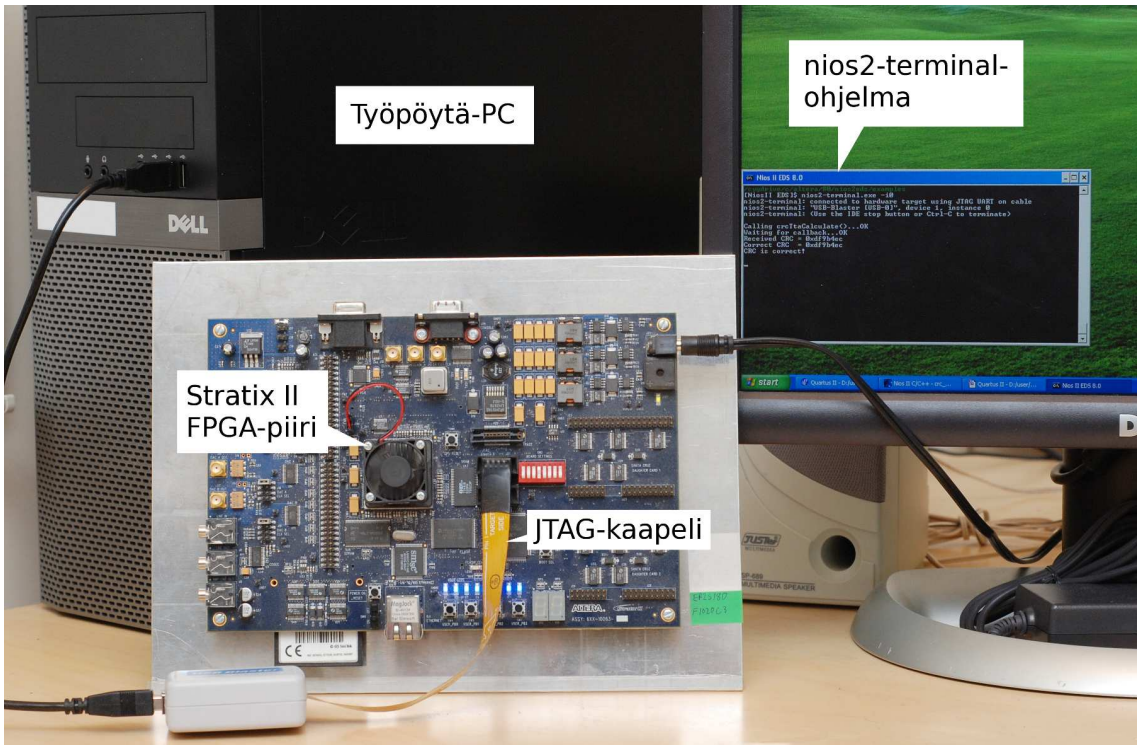
pitää lisätä Koskivuon komponenttikirjastoon graafisen Library Manager-työkalun avulla. Kactus-työkalulla voidaan graafisesti koota järjestelmä käyttämällä komponenttikirjastosta löytyviä komponentteja. Kuva 5.5 esittää Kactus-työkalussa tehtyä järjestelmää. Lopuksi Kactus-työkalulla generoidaan XML-kuvaus järjestelmästä.

Seuraavaksi käynnistetään KoskiGUI-työkalu, jonka avulla hoidetaan loppuosa Koskivuosta. Työkalun pääsyötteenä on Kactuksen generoima XML-kuvaus järjestelmästä, jonka mukaisesti työkalu tekee laitteistonkuvauskielisen toteutuksen järjestelmälle. Näiden lisäksi KoskiGUI:n avulla voidaan myös kääntää Nios II-prosessorilla suoritettava ohjelmakoodi ajettavaksi ohjelmaksi sekä generoida listaus synteesiin tarvittavista VHDL-tiedostoista ja FPGA:n pinnikytkennöistä.

Viimeisenä vaiheena on syntesoida koko järjestelmä ja suorittaa se FPGA:lla. Alkuvaiheessa sekä Nios:n että TTA:n ohjelmakoodin jätettiin debug-tulosteita, jotta toiminnan seuraaminen ja varmentaminen olisi helpompaa. Molemmilta prosessoreilta voidaan tulostaa JTAG UART-komponenttien kautta nios2-terminal-ohjelmaan. Käytännön testijärjestely on esitetty kuvassa 5.6. Pienen hienosäätämisen jälkeen järjestelmä saatiinkin toimimaan odotetulla tavalla, jolloin ylimääräiset tulosteet voitiin poistaa. Lopullisessa ohjelmassa Nios II laskee testidatalle CRC:n TTA:n avulla ja tulostaa saadun laskentatuloksen sekä oikean arvon CRC:lle, jonka avulla voidaan todeta järjestelmän toimivan. Koko järjestelmän resurssienkäyttö on myös listattuna taulukoon 5.4.

5.7 Lopputulos

Esimerkissä käytiin läpi sovelluskohtaisen TTA-prosessorin suunnitteluvaiheet ja sen liittäminen Koski-järjestelmänsuunnitteluvuohon. TCE-työkalut mahdollistavat nopean suunnittelusyklin. Esimerkin tapauksessa ohjelman kääntäminen TTA prosessorille ja ohjelman simulointi arkkitehtuurisimulaattorilla kestivät yhteensä 5-30 sekuntia tavallisella työpöytä-PC:llä. Vastaavasti prosessoritoteutuksen ja ohjelman



Kuva 5.6: Valokuva järjestelmän testaamisesta. Stratix II FPGA-lauta on kytketty JTAG-kaapelilla työpöytä-PC:hen. Järjestelmän JTAG UART-komponentit kommunikoivat JTAG-kaapelin avulla PC:llä suoritettavan nios2-terminal-ohjelman kanssa, jolloin järjestelmän testitulosteet saadaan näkyviin nios2-terminal-ohjelman ikkunaan.

muistikuvien generointi vei noin 10-15 sekuntia. Prosessorin suunnittelunopeus on siis lähinnä kiinni suunnittelijasta. Suunnitteluesimerkki osoitti myös, miten prosessorin käskykannan laajentaminen yksinkertaisella sovelluskohtaisella erikoisoperaatiolla voi nopeuttaa ohjelman suorittamista merkittävästi. Lisäksi vertailu Nios II-prosessoriin paljasti, että lähestulkoon samalla logiikkaelementtien käytöllä TTA-prosessori oli melkein kaksi kertaa nopeampi, jos Nios II:n käskykantalaajennusta hyödynnettiin. Ilman Nios II:n käskykantalaajennusta ero oli yli kymmenkertainen TTA:n hyväksi.

TTA-prosessorin liittäminen Koskivuohon onnistui helposti työssä toteutetulla Koski Integrator:lla. Koski Integrator hoitaa TTA-prosessorikomponentin liittämisen Koski-työkaluihin mahdollisimman automaattisesti, jolloin suunnittelijan ei tarvitse tehdä mekaanista käsityötä. Suuritöisin osuus yksittäisen TTA-prosessorin järjestelmäintegroinnissa oli HIBI-kommunikaation suunnittelu ja toteutus. Esimerkkisovelluksen tapauksessa kommunikaatio oli kuitenkin melko yksinkertainen toteuttaa.

Taulukko 5.4: Suunnitteluesimerkin synteesitulokset Stratix II DSP Pro FPGA-piirille. Alimmassa sarakkeessa on listattuna Stratix II FPGA-piirin tiedot resurssien osalta

	LE:t / kpl	**Muisti-bittiä**	f_{max} **/ MHz**	**Kertojat / kpl**
Nios II/f (crc)	1 563	-	130,34	8
minimaalinen TTA	1 005	76 288	116,08	0
paranneltu TTA	1 440	65 536	110,44	0
CRC TTA 1	1 506	41 344	112,70	0
CRC TTA 2	1 581	36 480	108,85	0
CRC TTA 3	3 101	46 080	113,82	0
Testijärjestelmä	13 846	224 512	82,84	8
Stratix II-piiri	143 520	9 383 040	500,00	768

Taulukko 5.5: Tietoja testatuista TTA-prosessoreista. Kellojaksolukemat eivät sisällä verifikaatioon kuluvaa aikaa. Viimeisen prosessorin suoritusaika riippuu ulkoisista tapahtumista, joten tarkkaa suoritusaikaa ei merkattu.

Arkkitehtuuri	**Suoritusaika kellojaksoina**	**Käskynleveys**	**Käskymäärä**
minimalistinen TTA	436 813	42	784
paranneltu TTA	56 452	126	161
CRC TTA 1	19 748	126	35
CRC TTA 2	19 748	50	35
CRC TTA 3	-	50	188

6. YHTEENVETO

Tässä työssä esiteltiin, miten siirtoliipaistuja prosessoreja suunnitellaan ja miten niitä voidaan hyödyntää järjestelmäsuunnittelussa FPGA-piireille käyttäen esimerkkinä Koski-järjestelmäsuunnitteluvuota. Siirtoliipaistu prosessoriarkkitehtuuri ei määrittele yhtä prosessoritoteutusta, vaan tarjoaa helposti räätälöitävän prosessorisuunnittelumallin. Tämän vuoksi soveltuvat erittäin hyvin käytettäväksi sovelluskohtaiseksi räätälöitävinä prosessoreina. Räätälöitävyyden lisäksi siirtoliipaistu prosessoriarkkitehtuuri kykenee hyödyntämään sovelluskoodista löytyvää käskytason rinnakkaisuutta, jonka ansiosta suorituskyky paranee skalaari-RISC arkkitehtuuriin nähden.

TCE-työkalut (TTA-based Codesign Environment) tarjoavat kehitysympäristön sovelluskohtaisille siirtoliipaistuille prosessoreille. TCE-työkaluilla prosessorisuunnittelu tapahtuu arkkitehtuuritasolla, jonka ansiosta suunnitteluiteraatiot ovat nopeita suorittaa. Suunnitteluvuon lopputuloksena saadaan joko FPGA:lle integroitu TTA-prosessori tai IP-komponentiksi käärityt TTA-prosessori, jota voidaan hyödyntää järjestelmäsuunnittelussa. Tässä työssä keskityttiin jälkimmäiseen tapaukseen käyttäen esimerkkinä Koski-suunnitteluvuota. Laitteistotason yhteesopivuuden takaamiseksi toteutettiin HIBI-liityntä TTA:lle soveltaen Nios II-prosessorille aiemmin tehtyä N2H2-lohkoa. N2H2-lohkon uudelleenkäyttäminen mahdollistaa myös Nios II-prosessorille suunniteltujen ajurikoodien hyödyntämisen, joka helpottaa merkittävästi sovelluksien siirrettävyyttä Nios II:n ja TTA:n välillä. Yhteensopivuus TCE- ja Koskityökalujen välillä saavutettiin toteuttamalla TCE-työkaluihin Koski Integrator-komponentti, joka käärii TTA-prosessorin IP-lohkoksi ja generoi sille Koski-työkaluihin yhteensopivan IP-XACT-kuvauksen.

Työssä käytiin esimerkin avulla lävitse, miten sovelluskohtaisen prosessorin suunnittelu- ja varmennusvuon vaiheet suoritetaan ja miten prosessori liitetään osaksi suurempaa järjestelmää. Suunnittelutyön ensimmäisessä vaiheessa räätälöitiin TTA-prosessori mahdollisimman sopivaksi esimerkkisovelluksen suorittamiseen. Räätälöinnissä muokattiin muun muassa prosessorin laskentayksiköitä, siirtoväyliä ja rekisteripankkia sekä toteutettiin käskykantaan sovelluskohtainen erikoisoperaatio. Toisessa vaiheessa suunniteltiin ja toteutettiin HIBI-kommunikaatio. Viimeisessä vaiheessa TTA-prosessori liitettiin muuhun järjestelmään ja järjestelmän toiminta varmennettiin. Yhteenveto työstä on koottu taulukkoon 6.1.

Taulukko 6.1: Yhteenveto työstä

Kohta	**Selite**
Työn tavoite	Hyödyntää TTA-prosessoreita järjestelmäsuunnittelussa
Prosessoriarkkitehtuuri	TTA
Suunnittelutyökalut	TCE
Järjestelmävuo	Koski
Toteutuskielet	C++, VHDL
Esimerkkitapaus	CRC-32-tiivisteen laskeminen
FPGA	Stratix II EP2S180F1020C3
Järjestelmän koko / LE	13 846
josta TTA:n osuus / LE	3 101
Järjestelmän kellotaajuus / MHz	50

LÄHTEET

[1] Wayne Wolf. *Modern VLSI Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2002.

[2] P. Lysaght, R. Chapman, and T. Durrani. System level integration, intellectual property, and the education of a new generation of system designers. In *Proc. Systems on a Chip, IEEE Colloquium on*, pages 2/1 –2/5, 1998.

[3] P. Rashinkar, P. Paterson, and L. Singh. *System-on-a-chip verification: methodology and techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[4] C. Edwards. Time softens logic. *Engineering Technology*, 5(5):33 –35, 2010.

[5] J.G. Tong, I.D.L. Anderson, and M.A.S. Khalid. Soft-core processors for embedded systems. In *Proc. Microelectronics, International Conference on*, pages 170 –173, 2006.

[6] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[7] Altera Corporation. *Nios II Processor Reference Handbook*, December 2010. `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`.

[8] Xilinx Inc. *MicroBlaze Processor Reference Guide*, November 2010. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/mb_ref_guide.pdf`.

[9] OpenCores. *OpenRISC 1200 Specification*, January 2011. `http://opencores.org/svnget,or1k?file=/trunk/or1200/doc/openrisc1200_spec.pdf`.

[10] D. Mattson and M. Christensson. Evaluation of synthesizable CPU cores. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, December 2004.

[11] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[12] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[13] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An FPGA-based VLIW processor with custom hardware execution. In *Proc. Field-programmable gate arrays, International Symposium on*, pages 107–117, 2005.

[14] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala. Impact of software bypassing on instruction level parallelism and register file traffic. In *Proc. Embedded Computer Systems: Architectures, Modeling, and Simulation, International Workshop on*, pages 23–32, 2008.

[15] O. Esko, P. Jääskeläinen, P. Huerta, C.S. de La Lama, J. Takala, and J.I. Martinez. Customized exposed datapath soft-core design flow with compiler support. In *Proc. Field Programmable Logic and Applications, International Conference on*, pages 217 –222, 2010.

[16] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proc. SPIE Multimedia on Mobile Devices*, pages 65070X–1 – 65070X–11, 2007.

[17] TCE: TTA-Based Codesign Environment. `http://tce.cs.tut.fi`.

[18] V-P. Jääskeläinen. Retargetable Compiler Backend for Transport Triggered Architectures. Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, March 2010. `http://tce.cs.tut.fi/`.

[19] P. Jääskeläinen. Instruction Set Simulator for Transport Triggered Architectures. Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, September 2005. `http://tce.cs.tut.fi/`.

[20] Mentor Graphics ModelSim: Advanced Simulation and Debugging. `http://model.com/`.

[21] GHDL: Open source VHDL simulator. `http://ghdl.free.fr/`.

[22] Altera Corporation. *Embedded Peripherals IP User Guide*, December 2010. `http://www.altera.com/literature/ug/ug_embedded_ip.pdf`.

[23] The LLVM Compiler Infrastructure Project. `http://llvm.org`.

[24] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Transactions on Embedded Computing Systems*, 5:281–320, 2006.

[25] The SPIRIT Consortium. *IP-XACT User Guide v1.2*, July 2006.

[26] T. Koskinen. Metadata-based Automated Configuration of System-on-Chip. Master's thesis, Department of Computer Systems, Tampere University of Technology, Tampere, Finland, June 2009.

[27] A. Kulmala. Multiprocessor system with general-purpose interconnection architecture on FPGA. Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, August 2005.

[28] Altera Corporation. *Avalon Interface Specification*, August 2010.

[29] IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks–Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Section One. *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pages c1 –597, 2008.

[30] Example implementation for calculating CRC. `http://www.netrino.com/code/crc.zip`.

[31] Tampere University of Technology, Department of Computer Systems. *TTA Codesign Environment v1.4 User Manual*, January 2011. `http://tce.cs.tut.fi/user_manual/TCE.pdf`.