Merja Maijanen

FPGA-PERUSTAISEN SOC:N VERIFIOINTI SUUNNITTELUN ERI VAIHEISSA

Insinöörityö
Kajaanin ammattikorkeakoulu
Tekniikan ja liikenteen ala
Tietotekniikan koulutusohjelma
Kevät 2005

Kajaanin ammattikorkeakoulu

INSINÖÖRITYÖ
TIIVISTELMÄ

Osasto	Koulutusohjelma
Tekniikka	Tietotekniikka
Tekijä(t) Merja Maijanen	
Työn nimi FPGA-perustaisen SoC:n verifiointi suunnittelun eri vaiheissa	
Vaihtoehtoiset ammattiopinnot Elektroniikan testaussuunnittelu	Ohjaaja(t) Asko Kinnunen
Aika 8.4.2005	Sivumäärä 80

Tiivistelmä

Insinöörityön tavoitteena oli tutkia FPGA-perustaisen järjestelmäpiirin (SoC, System-on-Chip) verifioimista suunnittelun eri vaiheissa. Työ oli osa SoC-projektia, jossa olivat mukana mm. Oulun yliopiston Mittalaitelaboratorio, Kajaanin ammattikorkeakoulu ja joukko yrityksiä. Projektin tavoitteena oli rakentaa SoC-osaamista, jota voitaisiin siirtää sulautettuja järjestelmiä tekevien yritysten käyttöön.

Työssä tutkittiin yleisiä verifiointimenetelmiä ja teoriaa kirjallisuuslähteiden perusteella. Tutustuttiin myös FPGA:lle toteutettavan sovelluksen suunnitteluvuohon ja eri verifiointimenetelmien käyttöön suunnittelun eri vaiheissa. Alteran valmistama FPGA-piiri (Stratix II) ja valmistajan oman Quartus II -suunnitteluohjelmiston verifiointityökalut olivat työssä erityisen tutkimuksen kohteena. Verifiointityökalujen ominaisuudet ja käyttö eri suunnitteluvaiheissa kuvattiin. Lyhyen esimerkin avulla tutustuttiin Quartus II -ohjelmiston verifiointityökalujen toimintaan ennen piirin toteutusvaihetta ja sen jälkeen.

Työ osoitti, että verifioinnin merkitys suunnittelun eri vaiheissa on suuri. Verifiointi vie yli puolet suunnitteluajasta ja sen toteuttaminen on usein työlästä. Verifiointitekniikoita on useita. Yleisimpiä ovat simulointi ja staattinen ajoitusanalyysi. Piirin toteutusvaiheen jälkeen voidaan käyttää sulautettua logiikka-analysaattoria virheiden löytämiseen. Alteran Quartus II -suunnitteluohjelmisto sisältää verifiointityökaluja FPGA:n suunnittelun eri vaiheisiin.

Luottamuksellinen

Kyllä
Ei X

Hakusanat

verifiointi, SoC, FPGA, Quartus II -ohjelmisto, verifiointityökalut, simulaatio, ajoitusanalyysi

Säilytyspaikka

Oulun yliopisto, Kajaanin kehittämiskeskus, Mittalaitelaboratorio, kirjasto
Kajaanin ammattikorkeakoulu, tekniikka, kirjasto

Kajaanin ammattikorkeakoulu
Kajaani Polytechnic

ABSTRACT
THESIS

Faculty	Degree programme
Faculty of Engineering	Information Technology
Author(s) Merja Maijanen	
Title Verification of an FPGA-based SoC at Various Design Levels	
Optional professional studies Design for Testability	Instructor(s) / Supervisor(s) Asko Kinnunen
Date 22.3.2005	Total number of pages 80

Abstract

The purpose of this Bachelor's thesis was to investigate how an FPGA-based System-on-Chip (SoC) is verified at various design levels. Firstly, common techniques and methodologies of SoC verification were studied. Secondly, the specific FPGA device (Stratix II) manufactured by Altera Corporation and its verification tools were investigated. The verification tools were part of the Altera's Quartus II design software for FPGA design.

The thesis was part of a research project. The aim of the project was to develop SoC expertise for companies working in embedded system technology. Among others, the Measurement and Sensor Laboratory of Oulu University and Kajaani Polytechnic took part in the project.

Common techniques and methodologies of SoC verification were found out in the literature. The flow of FPGA design was also under research to find out the purpose of different verification methods at various design levels. Altera's Quartus II design software for FPGA design and specially its verification tools were examined and tested. Short example software was performed to test Altera's verification tools before and after the implementation of the device.

The studies of the verification methods indicated that verification is a significant part of designing today's Soc devices. Verification is time-consuming and often very hard. Verification consists of several techniques. The most used methods are the simulation and the static timing analysis. The embedded logic analyzer is a tool to verify the device after the implementation. Altera's Quartus II design software provides the tools to verify the design in all design levels.

Confidential
Yes
No X

Keywords

verification, SoC, FPGA, Quartus II software, verification tools, simulation, timing analysis

Deposited at

Kajaani Polytechnic, Faculty of Engineering, library
Measurement and Sensor Laboratory, library

SISÄLLYS

SYMBOLILUETTELO

ALUT	Adaptive Lookup Table, mukautuva hakutaulukko
ALM	Adaptive Logic Module, mukautuva logiikkamoduuli, Stratix II -arkkitehtuurin logiikan peruslohko
AMS	Analog/Mixed Signal, analogia- ja sekasignaali
ASIC	Application Spesific Integrated Circuit, asiakaskohtainen piiri
EDA	Electronic Design Automation, elektroniikan suunnitteluautomaatio, esimerkiksi suunnittelutyökalut
FPGA	Field Programmable Gate Array, ohjelmoitava logiikkaverkko
HDL	Hardware Description Language, laitteistonkuvauskieli, laitteiston rakenteen tai toiminnan kuvaamiseen käytetty tekstikieli
ICE	In-Circuit Emulator, testattavaan laitteeseen kytkettävä mikroprosessoriemulaattori
I/O	Input/Output, sisääntulo/ulostulo
IP	Intellectual Property Block, virtuaalikomponentti, uudelleen käytettävä laitteistolohko
JTAG	Joint Test Action Group, integroitujen piirien testausliityntästandardi
LAB	Logic Array Block, logiikkaelementtien ryhmä
LE	Logical Element, logiikkaelementti
LUT	Lookup Table, hakutaulukko
RTL	Register Transfer Level, rekisterinsiirtotaso
SoC	System-on-a-Chip, järjestelmäpiiri
SRAM	Static Random Access Memory, muistityyppi
VHDL	Very high speed integrated circuit Hardware Description Language, laitteistonkuvauskielistandardi, yleinen menetelmä laitteistojen esitykseen

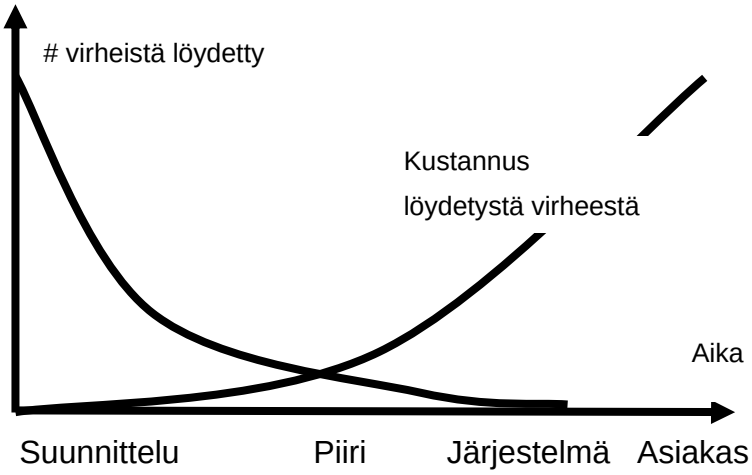
1 JOHDANTO

SoC:ssa (System-on-Chip, järjestelmäpiiri) integroidaan kaikki lopullisen tuotteen päätoiminnot yhteen piiriin. Tyypillinen SoC voi sisältää esimerkiksi prosessorin, väyliä ja niiden välisen sillan sekä useita oheislaitteita ja ohjaimia. Järjestelmäpiiri voidaan toteuttaa perinteisesti asiakaskohtaisena ASIC-toteutuksena, jolloin kyseessä ovat yleensä suuret valmistusmäärät. Ohjelmoitavien logiikkaverkkojen (FPGA) tiheydet ovat kuitenkin kasvaneet ja niiden käyttö järjestelmäpiireinä on lisääntynyt. FPGA-tyyppisten toteutusten etuna on järjestelmän uudelleenohjelmoitavuus ja muunneltavuus.

Suunniteltaessa asiakaskohtaisia ASIC-piirejä tai käytettäessä FPGA-piirejä on verifiointi, suunnitelman toiminnan varmentaminen, olennainen osa suunnitteluprojektia. Verifioinnin osuus suunnittelussa on myös tärkeää, jotta SoC-tuote saadaan nopeasti tuotantoon ja markkinoille. FPGA-piirien monimutkaisuus lisääntyy, piireille integroidun logiikan määrä kasvaa ja piirien koot pienenevät koko ajan Mooren lain vauhtia (logiikkaportteja, transistoreita, voidaan integroida yhdelle piirille kaksinkertainen määrä joka 18. kuukausi). Tämä asettaa suunnittelun verifioinnille ja käytetyille työkaluille haasteita. Voidaanhan yhteen piiriin sisällyttää yhä enemmän toimintoja ja logiikkaa. Piirin sisäisten yhteyksien aiheuttamat viiveet käyvät yhä merkityksellisemmiksi, ja ne täytyy pystyä arvioimaan tarkasti, jos halutaan saavuttaa ajoitukseen liittyvät tavoitteet. Piirien kapasiteettien kasvaessa ja sovelluksien paisuessa yhä suuremmiksi täytyy myös ohjelmointitapoja kehittää esimerkiksi lohkojen ja niiden uusiokäytön avulla. Piirin fyysiset ominaisuudet on otettava huomioon suunnittelussa. Metallikerrosten lisääntyminen, käyttöjännitteiden aleneminen, miljoonien osasten sijaitseminen yhdessä piirissä, korkeat kellotaajuudet ja muut vastaavat seikat aiheuttavat huomioitavia seikkoja signaalien käyttäytymisessä sekä myös suunnittelussa. [1.]

SoC-järjestelmän verifioinnin osuus koko suunnittelutyöstä on 40 - 70 %. Verifioinnin yhteydessä on aina mietittävä, kuinka paljon verifioidaan, mikä riittää, ja kuinka paljon aikaa siihen voidaan käyttää [1, s. 7]. Verifioidaanko tuotetta niin

kauan, kun rahat riittävät tai kunnes tuote lähtee markkinoille? Onko viikko riittävä aika, jos virheitä ei siihen mennessä ole löytynyt? Onko HDL-koodin jokainen rivi käytävä läpi? Tyypillisesti pääosa virheistä löytyy ensimmäisten viikkojen aikana. Virheiden löytyminen myöhemmässä vaiheessa on kallista (kuva 1). [2.]



Kuva 1. Mitä myöhemmin suunnitteluvirhe löydetään, sitä suurempia kustannuksia siitä aiheutuu. [3]

On myös ratkaistava, missä vaiheessa verifiointia suoritetaan. On pohdittava, millaisia strategioita ja teknisiä valintoja verifioinnissa käytetään. Tämä asettaa haasteita sekä verifioinnin suorittajille että verifiointiratkaisujen tarjoajille. Verifiointiin käytettäviä teknisiä vaihtoehtoja on tarjolla runsaasti. Ne voidaan yleisesti jakaa neljään eri ryhmään: simulaatiopohjaiset, staattiset ja formaalit tekniikat sekä fyysinen verifiointi ja analyysi. [1, s. 7.]

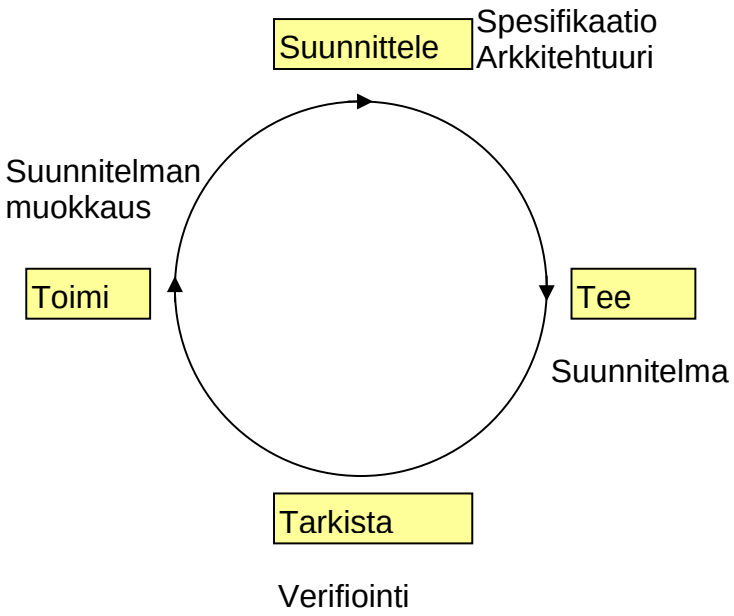
System-on-Chip-osaamista rakennetaan tutkimushankkeessa, jossa mukana ovat Oulun yliopiston Mittalaitelaboratorio, VTT Elektroniikka, Kajaanin ammattikorkeakoulu sekä joukko yrityksiä. SoC-osaamista siirretään sulautettuja järjestelmiä tekevien yritysten tuotekehityskäyttöön. Tämä insinöörityö on osa työpakettia, jossa tutkitaan SoC-teknologian käytettävyyttä tuotteissa. Työssä tarkastellaan FPGA-piirille toteutettavan SoC-sovelluksen suunnittelun varmentamista piirisuunnittelun eri vaiheissa. Kohteena ovat erityisesti Alteran valmistama Stratix II -piiri ja piirinvalmistajan oman Quartus II -suunnitteluohjelmiston verifiointimenetelmät.

Työssä tarkastellaan yleisiä SoC:n verifiointiin käytettyjä tekniikoita piirisuunnittelun eri vaiheissa. Ensin esitellään yleisiä verifiointiin liittyviä seikkoja, verifiointisuunnitelma ja FPGA:n suunnitteluprosessi. Seuraavaksi kuvataan lyhyesti yleiset SoC:n verifiointitekniikat piirisuunnitteluvaiheessa ja korttitasolla sekä erilaiset lähestymistavat SoC-verifiointiin. Alteran FPGA-piirin verifiointia käsitellään Quartus II -ohjelmiston ja sen sisältämien verifiointityökalujen avulla. Myös JTAG-menetelmän tarjoamat mahdollisuudet Stratix II -piirille esitellään. Esimerkkisuunnitelman avulla toteutetaan suunnittelun verifiointia Quartus II -ohjelmiston avulla.

2 SOC:N VERIFIOINTI

2.1 Yleiskatsaus verifiointiin

Verifiointi on prosessi, jota käytetään osoittamaan suunnitelman toiminnallinen paikkansapitävyys ja virheettömyys. Verifiointi eroaa testauksesta siten, että testauksessa varmistetaan valmistetun laitteen virheettömyys. Testausta voidaan suorittaa jokaiselle valmistetulle laitteelle, kun taas verifiointia tehdään yhden kerran laitteelle sen suunnitteluvaiheessa. Testaus liittyy laitteen rakenteellisuuteen, verifiointi taas toiminnallisuuteen. Verifioinnin tarkoitus on varmistaa, että jonkin muutoksen tulos on sellainen, joka oli odotettavissa. Verifiointia suoritetaan simulaation, laitteistoemulaation ja prototyyppien tai staattisten ja formaalien metodien avulla ja sen avulla vastataan suunnittelun laadusta. Kuvassa 2 on esitetty verifioinnin periaate. [1.]



Kuva 2. Verifioinnin avulla varmennetaan määrittelyjen mukaisen suunnittelun virheettömyys. [3]

Suunnittelussa on erilaisia abstraktiotasoja. Verifioinnin päämäärä on varmistaa, että suunnittelu vastaa toiminnallisia ja ajoituksellisia vaatimuksia jokaisella abstraktion tasolla. Kun luodaan suunnittelu korkeammalla abstraktiotasolla, ve-

rifioidaan se tällä tasolla. Suunnittelu käännetään alemmalle tasolle ja verifioidaan, että se on johdonmukainen edellisen, korkeamman tason kanssa. Seuraavaksi se verifioidaan ko. abstraktiotasolla. Näin jatketaan, kunnes on saavutettu alin abstraktiotaso. [4.]

Verifiointi on yksinkertaista, kun kyseessä on vaikkapa yksi flip-flop-kytkentä. Tiloja on kaksi ja testijonoja vaaditaan neljä. Jos kyseessä onkin 5000 portin mikroprosessori, jossa on 208 rekisteribittiä ja 13 varsinaista sisääntuloa, on mahdollisia tilamuutoksia $2^{bits+inputs} = 2^{221}$ kappaletta. Jos kyseessä on piiri, jossa on miljoonia portteja, ovat verifioinnin haasteet melkoiset. Suunnittelut ovatkin usein niin suuria, ettei täydellistä toiminnallista kattavuutta voida varmistaa. Verifioinnin avulla voidaan osoittaa virheet, mutta ei niiden puutetta. [5.][6.]

Vaikka SoC:ien verifiointi on hyvin samankaltaista kuin perinteisten ASIC-piirien, on niiden verifioinnissa kuitenkin seikkoja, jotka asettavat verifioinnille erityishaasteita. Eri komponenttien välinen integraatio on tarkastettava ja useiden alijärjestelmien keskinäiset riippuvuussuhteet voivat olla erittäin monimutkaisia. Suunnitelmat ovat usein suuria, mutta ne on toteutettu lyhyellä ajalla pienehköjen suunnittelutiimien avulla. IP-lohkoja ja prosessoreita käytetään paljon. Usein syvällistä ymmärrystä koko suunnitelmasta ei ole saavutettu, kun verifiointia suoritetaan. [7.]

Eräs suunnittelua nopeuttava ominaisuus on aiemmin suunniteltujen lohkojen käyttäminen. Lohkoja kutsutaan joko virtuaalikomponenteiksi tai IP-lohkoiksi (Intellectual Property Block). IP-lohkot voivat olla omaa, piirivalmistajan tai kolmannen osapuolen suunnittelemaa logiikkaa, joka on rajattu toiminnallisiin lohkoihin [8]. Lohkoja voidaan käyttää suunnittelussa, jolloin uuden suunnittelun määrä vähenee. Lohkot on useimmiten aiemmin verifioitu, joten suunnittelumäärän lisäksi myös verifioinnin tarve vähenee. [1, s. 2] Vaikka valmistaja on verifioinut IP-lohkon, on käyttäjän kuitenkin verifioitava se omassa ympäristössään. Lohkot ovat usein ”mustia laatikoita” ja niiden sisältöä on vaikea nähdä, eikä simulaatiomalleja niille ole aina saatavissa. Virheiden haku voi olla vaikeampaa, sillä ei voida aina tietää, onko virhe lohkon sisällä vai sen liitynnässä. [9.][7.]

Laitteiston ja ohjelmiston suhde on otettava koko ajan huomioon ja pyrittävä siihen, että ne yhdessä muodostavat testattavan laitteen. SoC-suunnitteluissa on tyypillisiä alueita, joissa vikatilanteita esiintyy, kuten esimerkiksi vuorovaikutus lohkojen välillä tai jaettuihin resursseihin pääsyn ristiriitatilanteet. [10.]

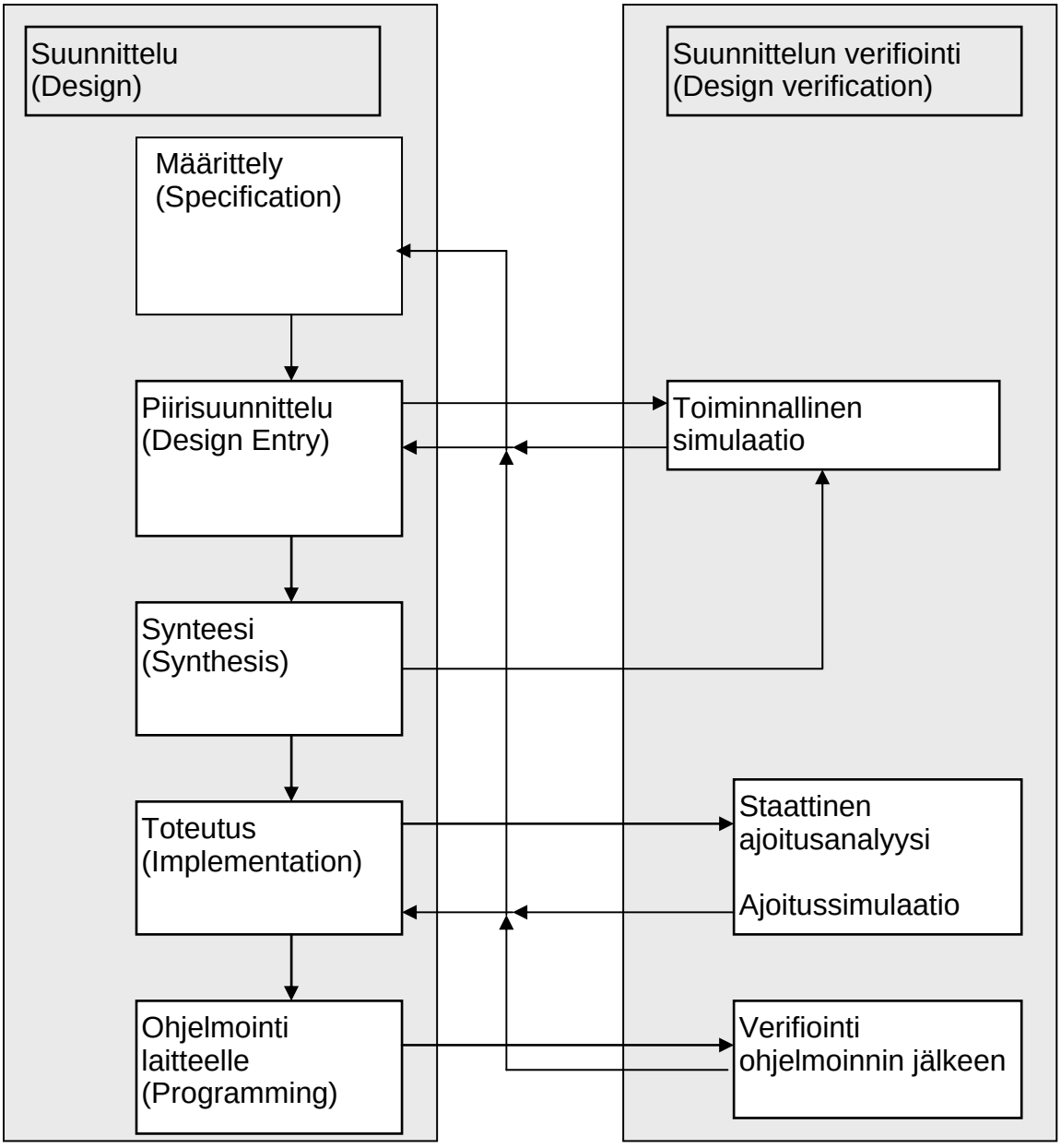
2.2 FPGA:n suunnitteluprosessi

Ohjelmoitavalle logiikkaverkolle, FPGA:lle, tehtävä sovellus toteutetaan käytännössä tietokoneella suunnitteluohjelmistojen avulla. Suunnittelun aluksi luodaan määrittelyt, spesifikaatiot, joiden tehtävä on kuvata piirin toimintaa. Spesifikaatio on tavallisesti tekstidokumentti. Määrittelyn jälkeen tehdään piirisuunnittelu (Design Entry), joka on hierarkkinen prosessi. Ensin laaditaan piirin päälohkokaavio, jota voidaan syventää tarvittaessa. Lohkojen sisällöt kuvataan piirikaaviona, tilakaaviona, korkean tason kuvauskielisenä kuvauksena tai valmislohkona. Suunnittelutyökalun avulla lohkojen kuvaukset yhdistetään ja niistä muodostuu yhteinen tiedosto. Toiminnallisen simuloinnin avulla simuloidaan lohkoja ja koko piiriä ja selvitetään, vastaako niiden toiminta spesifikaatiossa määriteltyä toimintaa. Simulointi tehdään simulaattoriohjelmalla. Viiveet ja muut ajoitusparametrit jätetään vielä ottamatta huomioon eikä niitä tiedetä. [11.]

Synteesi on seuraava vaihe piirisuunnittelun jälkeen. Suunnitteluohjelma optimoi piirisuunnittelun tuloksen ja poistaa siitä tarpeettomat komponentit. Synteesi on yleisluontoinen kuvaus, eikä siinä vielä soviteta suunnittelua minkään todellisen piirin tai piiriperheen arkkitehtuuriin. Synteesissä käytetty korkean tason kieli, esimerkiksi VHDL tai Verilog, käännetään vastaamaan piirin perusalkioita. Tuloksena on kytkentälista (netlist). Synteesityökalu on tavallaan kuten kääntäjä, jolla korkean tason kieli, kuten C-kieli, käännetään sarjaksi primitiivikomentoja, jotka voidaan sitten suorittaa prosessorilla [12]. Synteesivaihe on raskas varsinkin mutkikkaille piireille ja vaatii usein aikaa. Synteesin jälkeen voidaan suorittaa uusi toiminnallinen simulointi. Jos virhetilanteita esiintyy eikä kuvaus toimikaan kuten oli tarkoitus, palataan takaisin piirisuunnitteluvaiheeseen tekemään tarvittavat korjaukset. [12.] [11.]

Toteutus- eli implementaatiovaiheessa valitaan käytettävä piirityyppi tai -perhe. Myös erilaisille suunnitteluparametreille voidaan määritellä arvoja, joita ohjelma pyrkii noudattamaan toteuttaessaan synteesin tuloksen piirille. Parametreja ovat esimerkiksi haluttu toimintanopeus, piirin pinta-ala ja määrättyjen lohkojen sijoitus piiriin. Seuraavaksi ohjelma sijoittaa suunnitelman piirille ja määrittelee kytkentämatriisin kytkennät. Jos suunnitelma ei mahdu yhteen piiriin, jakaa ohjelma sen useammalle piirille ja määrittelee niiden väliset kytkennät. Tässä vaiheessa tehdään ajoitusanalyysi. Sen avulla voidaan tutkia piirin ajoitusta ja niissä mahdollisesti piileviä ongelmia. Kun suunnittelu on sijoitettu piiriin, tiedetään piirin sisäiset viiveet. Voidaan tehdä ajoitussimulaatiota simulointiohjelmalla. Se perustuu implementaatiossa generoitujen tiedostojen ajoitustietoihin [10]. Jos viiveet muuttavat piirin toimintaa siten, että toiminta ei olekaan odotettua, voidaan muuttaa suunnitteluparametreja tai suunnittelua. Implementaatiota kutsutaan usein suunnitelman sijoitus ja reititys -toiminnaksi (eri yhteyksissä esiintyy useimmiten nimellä place-and-route). Kokonaisuudessaan se on kuitenkin prosessi, jossa sijoituksen ja reitityksen lisäksi myös käännetään, kartoitetaan looginen suunnittelu kohteeseen ja generoidaan varsinaista ohjelmointia varten tarvittava bittitiedosto [12.][11.]

Kun ajoitussimulaation avulla voidaan todeta piirin toimivan halutulla tavalla, voidaan suorittaa itse piirin ohjelmointi. SRAM-perustaisen FPGA:n tapauksessa ohjelmoidaan piirin ohjelman sisältävä kiintomuisti. Ohjelmoinnin jälkeen voidaan testata piirin toimintaa. Usein rakennetaan laitteen prototyyppi ja käytetään logiikka-analysaattoria. Piirin sisälle voidaan myös ohjelmoida testipisteitä, joista voidaan ohjelman avulla seurata piirin toimintaa. Piiriä siis käytetään osittain testaamaan itseään. [11] Kuvassa 3 on esitetty tyypillisen suunnitteluvuon kuvaus, kun halutaan tehdä FPGA:n avulla toteutettavia suunnitelmia. [12.]



Kuva 3. Tyypillinen suunnitteluvuo käytettäessä FPGA:ta [12].

Suunnittelutyökalut

FPGA:lle toteutettavat suunnitelmat tehdään suunnittelutyökalujen avulla. Ne ovat ohjelmia, joita voidaan joko saada piirinvalmistajalta tai riippumattomilta ohjelmistotoimittajilta. Piirinvalmistajien ohjelmistot ovat halvempia tai jopa ilmaisia määrätyillä ominaisuuksilla, piirinvalmistajista riippumattomien ohjelmistojen avulla voidaan taas suunnitella useiden eri valmistajien piirejä. SoC-tutkimushankkeessa käytetylle Alteran valmistamalle Stratix II -piiriperheen

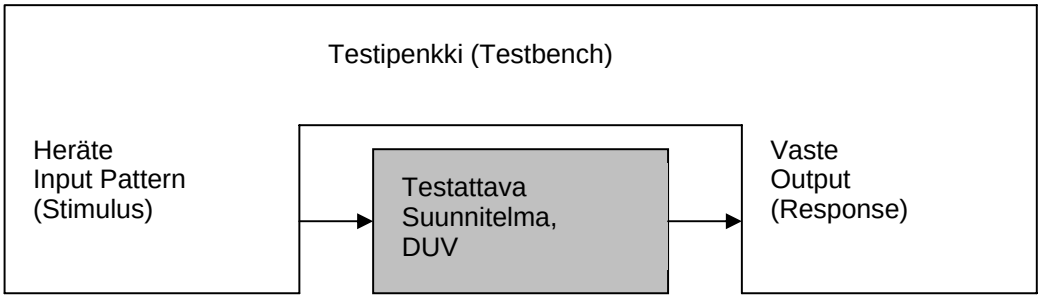
FPGA:lle voidaan käyttää piirinvalmistajan omaa Quartus II -suunnittelu-ohjelmistoa. Ohjelmisto sisältää suunnitteluun ja verifiointiin tarvittavat osat. Sen ominaisuuksien (SOPC Builder) avulla on myös mahdollista määritellä piirissä käytetyt komponentit, kuten erilaiset valmislohkot, prosessoriytimet, muistit jne. Ohjelmistojen lisäksi tarvitaan työkaluja, kuten ohjelmointilaitteet ja -kaapelit. Ohjelmoidun Stratix II -piirin testaukseen ja kokeiluun on saatavilla myös kokeilulevy (development board). Levyllä on asennettuna käytettävä FPGA, erilaisia liitäntöjä sekä muita komponentteja. Usein kokeilulevyllä on teholähde, muistipiirejä, kello, kytkimiä, ledejä ja liitäntöinä tehonsyöttöliitäntä, FPGA:n ohjelmointiliitäntä, videoliitäntä, USB-liitännät ja muita vastaavia osia. [11.] [13.]

2.3 Verifiointisuunnitelma

Verifiointisuunnitelma, joka toteutetaan tuotteen spesifikaation pohjalta, on yksi suunnittelun raporteista. Spesifikaatiossa kuvataan toteutettavat ominaisuudet, olosuhteet, missä ne tapahtuvat ja mitkä ovat odotetut tuotokset. Toteuttamismääräyksiä spesifikaatio ei kuitenkaan määritä, vaan toteuttaminen jätetään rekisterinsiirtotason (RTL) suunnittelijoille. Verifiointisuunnitelman avulla varmistetaan jokaisessa tuotteen kehitysvaiheessa, että suunnitelma toimii oikein ja sen suorituskyky täyttää ajoitusvaatimukset. Testisuunnitelman toteuttamisesta vastaavat sekä RTL-suunnittelijat että verifiointisuunnittelijat yhdessä [14]. Verifiointisuunnitelma sisältää mm. testistrategiat korkean tason moduulille ja eri lohkoille, vaadittavat verifiointityökalut ja selkeät kriteerit, joiden perusteella voidaan päättää, onko verifiointi tuloksellisesti valmis. [2.]

Verifiointisuunnitelmaa tehtäessä tunnistetaan suunnittelun keskeiset toiminnot. Kuvataan tilanteet, joissa avainominaisuudet sekä niiden odotetut vasteet tulevat esiin. Määritellään myös paljonko ja kuinka monimutkaisia testipenkkejä tarvitaan. Testipenkit kirjoitetaan, jotta vahvistetaan oikeaksi spesifikaatio, ei todentamaan suunnittelun toteuttamista, implementaatiota. Jokainen ristiriita suunnittelun toteuttamisessa verrattuna testipenkin tuloksiin on selvennettävä siten, että spesifikaatio on pohjana. [14.]

Testipenkki (testbench), on verifiointiympäristö, joka on kehitetty lisäämään herätteitä (stimulus) testattavaan laitteeseen (DUV, design under verification). Se myös määrittelee onko ulostuleva vaste (response) odotetun kaltainen. Testipenkki on usein toteutettu käyttämällä VHDL- tai Verilog -kieliä. Kuvassa 4 on esitetty testipenkin yksinkertaistettu lohkokaavio. [5.]



Kuva 4. Testipenkki (testbench) [2]

Verifiointisuunnitelman yhteydessä on tarpeellista yksilöidä saatavilla olevat ja käytettävät verifiointityökalut (simulaattorit, testipenkkityökalut jne.). Myös ulkoisten liitäntöjen ominaisuudet (heräte ja vaste, ajoitusvaatimukset, Read vs. Write -toiminnot) on tunnistettava. Saatavilla voi olla HDL-malleja avuksi testipenkin kehittämiseen. Valmiiksi suunnitellut IP-lohkot, joita suunnitelmassa käytetään, ovat usein saatavilla valmiiksi pakattuine HDL-testipenkkeineen. Suorituskyvyn vaatimusten lisäksi on tarpeellista myös tunnistaa toiminnallinen vuorovaikutus (mitkä sisäiset liitynnät ovat vuorovaikutuksessa ja onko se tutkittavissa ulkoisten liitäntöjen kautta.) Lisäksi on huolehdittava IP-moduulien integroimisesta järjestelmään ja varmistettava niiden liitäntöjen toimivuus [1]. [14.]

Verifioinnin toimeenpanotasolla suoritetaan sekä simulaatiota että laitteen korttitasolla (in-system) tapahtuvaa testausta. Jotta verifiointisuunnitelma olisi kehitetty juuri omalle suunnittelulle sopivaksi, on hyvä, että on eroteltu välttämättä simulointia tarvitsevat toiminnot niistä, jotka voidaan testata korttitasolla. Esimerkiksi pääliitynnät olisi testattava simuloinnin aikana, mutta testataanko ne kaikki? Simulointi on oleellista, kun valmistaudutaan laitteen kortilla tapahtuvaan testaukseen. Jos toiminnallisia liitäntöjä ei ole osoitettu kelvollisiksi, aiheuttaa se ongelmia korttitason testauksessa. On tarpeellista tietää, mitä toimintoja ja liitäntöjä tarvitaan, jotta korttitason testaus voidaan aloittaa. Ennen kuin virheiden haku ja korjaus (debuggaus) voidaan aloittaa korttitasolla, on esimerkiksi saa-

tava suoritettua käskyjä mikroprosessilla. Ennen kuin koodi voidaan suorittaa, on se ladattava PCI-väylältä SDRAM-muistiin. Vaiheiden oikean toiminnan ja ajoituksen on kuitenkin oltava kunnossa, jotta suunnitelman debuggaus voidaan aloittaa korttitasolla. Liityntäyksiköt onkin ensin simuloitava moduuleina ja sen jälkeen osana järjestelmää. [14.]

Verifiointisuunnitelman etuja ovat testaukseen liittyvän ympäristön kehittäminen aikaisessa vaiheessa ja rinnakkain muun suunnittelun kanssa. Verifioinnista aiheutuu tuotteen suunnittelussa kuormitusta ja se on otettava huomioon tuotteen toimitusajassa. [2.]

3 SOC:N VERIFIOINNIN TEKNIIKAT JA MENETELMÄT

3.1 Verifiointitekniikat

Verifiointitekniikoita on useita. Ne voidaan jakaa karkeasti neljään luokkaan: simulointipohjaiset tekniikat, staattiset tekniikat, formaaliset tekniikat sekä fyysinen verifiointi ja analyysi. Parhaan SoC:n verifioinnin varmistaa erilaisten menetelmien yhdistelmä. [1, s. 7.]

3.1.1 Simulaatiopohjaiset tekniikat

SoC:n suunnittelussa suunnittelun toiminnallinen verifiointi, simulointi, vie suuren osan suunnitteluajasta. Simulointi ei kuitenkaan ole sama asia kuin verifiointi. Verifiointi on toimintasuunnitelma, jonka avulla varmistetaan, että kaikki järjestelmän ominaisuudet vastaavat sille asetettuja spesifikaatioita, simulointi taas on verifioinnissa käytetty työkalu. [14.]

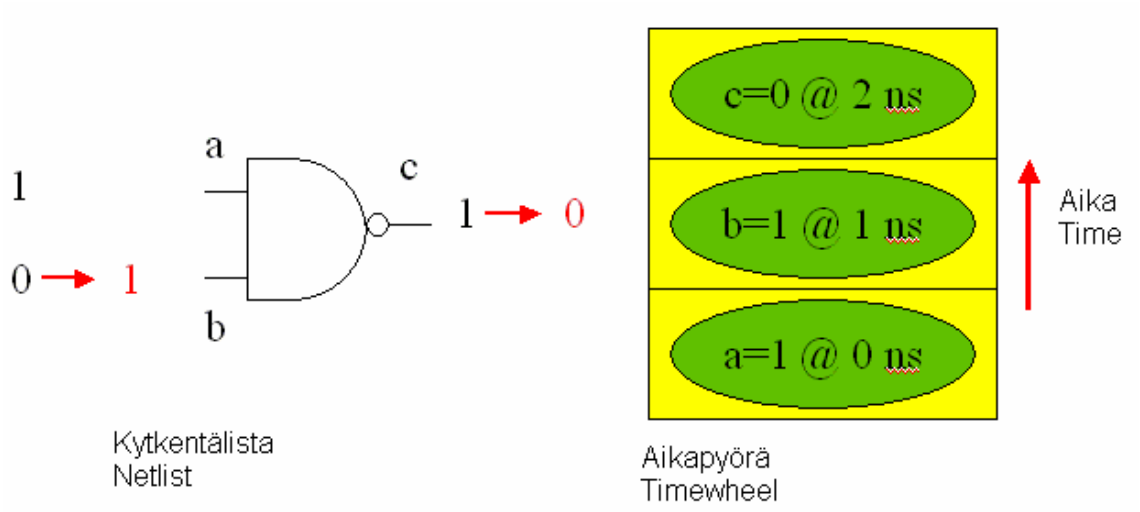
Suunnitelmien monimutkaisuus ja suunnitteluajan vaatimukset ovat kasvaneet ja tuote halutaan markkinoille nopeasti. Integraatioaste kasvaa ja yhä enemmän toimintoja voidaan lisätä yhdelle piirille. Tällöin myös simuloinnin osuus kasvaa. Simulointia käytetäänkin suunnittelun useissa vaiheissa. Varhaisessa vaiheessa korkean tason simuloinnin avulla tehdään analyysiä ja ennakoidaan suorituskykyä, keskivaiheella suunnittelun simulointia voidaan käyttää ohjelman algoritmien kehittämiseen ja laitteen jalostamiseen. Myöhemmissä suunnitteluvaiheissa voidaan simulointia käyttää varmistamaan, että suorituskyvylle asetetut vaatimukset on saavutettu ja laitteisto ja ohjelmisto toimivat oikein. On myös tärkeää, että simulointia voidaan käyttää siten, että se mahdollistaa ohjelmiston ja laitteiston rinnakkaisen kehityksen. Näin säästetään suunnittelun kokonaisaikaa. [15.]

Simulaation avulla verifioidaan suunnitelman toiminnallisia ja ajoituksellisia ominaisuuksia kaikilla abstraktio- ja esitystasoilla: RTL-tasolla, porttitasolla sekä transistoritasolla. Yleisimmät simulaatiotekniikat ovat tapahtumapohjainen ja jaksopohjainen simulaatio. Ne ovat molemmat HDL-pohjaisia, jolloin suunnittelu

ja testipenkki kuvataan HDL:n avulla. Muita simulointimenetelmiä ovat tapahtumapohjainen verifiointi, koodin kattavuus, AMS-simulaatio, HW/SW-yhteisverifiointi, emulointi, nopeat prototyypit, laitteiston mallintajat ja laitteisto-kiihdyttimet. [1.]

Tapahtumapohjaiset simulaattorit

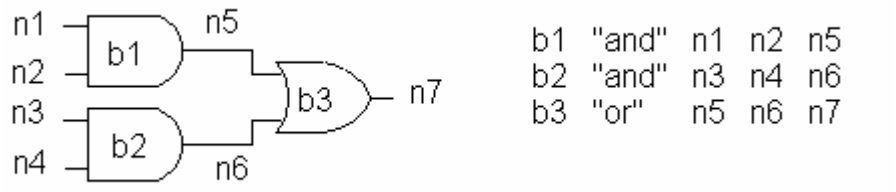
Eniten käytetty simulaatiotekniikka on tapahtumapohjainen simulaatio (event-based simulation), joka suorittaa sekä ajoituksellista että toiminnallista verifiointia. Tapahtumapohjaisten simulaattoreiden toiminta perustuu tapahtumiin, yhteen kerrallaan, ja niiden etenemiseen läpi suunnittelun kunnes on saavutettu pysyvä tila. Tapahtuma on (signaalin) muutos verkon (net) arvossa määrätyllä ajan hetkellä. Kun jokin signaali vaihtaa tilaa, on laskettava kaikkien niiden signaalien tilat, joihin se vaikuttaa. Simulaatio-ohjelma pitää lukua tapahtumista aikapyörän (time wheel) avulla, jonne tapahtumat talletetaan aikajärjestyksessä. Pyörää ”pyöritetään” eteenpäin seuraavaan aikapaikkaan, jossa on tapahtuma, ja simulointi aloitetaan sieltä (kuva 5). [16.]



Kuva 5. Tapahtumapohjainen simulaatio. Tapahtuma on muutos yhtymäkohdan loogisessa arvossa tietyllä ajan hetkellä. Tapahtumat talletetaan aikapyörään. [17]

Tapahtuman johdosta simulaattori arvioi uudelleen signaalien tilat ja laskee uuden ulostulon. Ulostulot voivat toimia seuraavan kohteen sisääntuloina. Uudelleen arviointia tapahtuu, kunnes muutoksia ei enää leviä suunnittelun läpi. [18.]

Kytkentälista (netlist) on keskeisen datan, ”avaindatan”, esitys suunnittelussa. Siinä luetteloidaan komponentit ja yhdistetään ne solmukohtien (nodes) avulla (kuva 6).



Kuva 6. Kytkentälistan (netlist) komponentit ja solmukohdat (n1 - n7) [19]

Kytkentälistaa tarvitaan simulaatiota ja implementaatiota varten. Se voi esittää esimerkiksi transistori- tai porttitasoa. Kytkentälistoja käytetään suunnittelun eri vaiheissa. [19.]

Tapahtumapohjainen simulaatio on tehokas ja tarkka, myös ajoituksen suhteen, ja sopii kaikentyyppisiin suunnitteluihin. Koska se on tapahtumavetoinen tekniikka, se huomioi vain aktiiviset solmukohdat, mikä saa aikaan tehokkuuden. Sen avulla löydetään lyhytkestoiset ongelmatilanteet ja asetusaika- ja pitoaikarikkomukset. Sen nopeus on riippuvainen suunnittelun koosta sekä aktiivisuustasosta simulaatiossa. Suuressa suunnitelmassa voi nopeus olla hidaskin, sillä tapahtumia arvioivat algoritmit ovat monimutkaisia ja ulostulot arvioidaan useita kertoja. [1, s. 8.] [4.]

Teollisuudessa esiintyy kahdentyyppisiä tapahtumapohjaisia simulointityökaluja. Toinen simulaattorityyppi kääntää ohjelmakoodin tietokoneen ymmärtämäksi ohjelmaksi (compiled-code simulator). Se hyväksyy HDL-kuvatun suunnittelun, kääntää sen datarakenteeksi ja ajaa saatua suoritettavaa ohjelmaa isäntäkoneessa. Esimerkkeinä simulaattoreista ovat Cadencen NC-Verilog ja VCS (Verilog Compiled Simulation). Toinen simulaattorityyppi tulkkaa koodin (interpreted-code simulator) jokaisen rivin ja ajaa sitä isäntäkoneella. Se myös hyväksyy HDL:n avulla kuvatun suunnittelun. Esimerkkinä työkalusta on Cadencen Verilog-XL [20].

Kun suunnittelulle valitaan sopivaa tapahtumapohjaista simulointiratkaisua, on otettava huomioon työkalun riittävä kyky käsitellä suunnittelun monimutkaisuutta, isäntäkoneen vaatima muisti ja käännösaika. Työkalun suorituskyky on tärkeimpiä ominaisuuksia. Suorituskykyyn vaikuttavat mm. suunnittelun koko, testivektoreiden määrä, muistin käyttö ja muut vastaavat seikat. Työkalun pitäisi myös kyetä käsittelemään mahdollisimman monia standardien määrittelemiä kielten ominaisuuksia. Virheiden löytämisen ja korjauksen ominaisuudet työkalussa nopeuttavat vikojen löytymistä. Simulaation tuottama informaatio voidaan esittää aaltomuotona tai raportteina. Simulaattorilla olisi myös oltava hyvä liityntä esim. laitteistokiihdyttimiin ja emulaattoreihin. Tukipalvelut työkalun käyttäjälle kannattaa huomioida. [1.]

Jaksoperusteiset simulaattorit

Jaksoperusteiset simulaattorit (cycle-based simulators) laskevat ja havainnoivat jokaisen signaalin yhden kerran kellojaksoa kohden. Signaalin tilaan vaikuttava tulo otetaan siis huomioon, vaikka se ei olisikaan vaihtunut edellisen kellojakson jälkeen. Simulaatiotekniikka perustuu kellojaksoihin, ei tapahtumiin. Niiden ainoa ”tapahtuma” on kellon aktiivinen reuna [18]. Jaksoperusteisia simulaattoreita voidaan käyttää vain synkronisessa eli tahdistetussa logiikassa (tilaelementit vaihtavat arvoa kellon aktiivisella reunalla) [4]. Ne ovat huomattavasti nopeampia kuin tapahtumapohjaiset simulaattorit, mutta simulaatiovirheitä voi esiintyä. Ne sopivat parhaiten suuriin suunnitteluihin, kuten mikroprosessorit, asiakaskohtaiset piirit ja SoC:it. Käyttöä rajoittaa vaatimus synkronisuudesta, sillä harvat reaalimaailman piirit ovat täysin synkronisia [18]. Jaksoperusteiset simulaattorit eivät anna informaatiota viiveistä. Ne eivät myöskään löydä lyhytkestoisia virhetilanteita koska ne vastaavat vain kellosignaaliin [4]. Koska ne eivät kiinnitä huomiota ajoitukseen, on käytettävä ajoituksen verifiointiin staattista ajoitusanalyysityökalua (static-timing analysis) [5, s. 8].

Transaktiopohjainen verifiointi

Transaktiopohjaisen verifioinnin avulla voidaan kohottaa verifiointiympäristön abstraktiotasoa, sillä signaalitason verifioinnista voidaan siirtyä tapahtumatasolle. Transaktio (tapahtuma) on datayksikkö, kuten esimerkiksi

Ethernet-paketti tai MPEG-kehys. Kaikki mahdolliset tapahtumatyypit suunnitelmaan sisältyvien eri lohkojen välillä luodaan ja testataan systemaattisesti. Tekniikka kohottaa verifioinnin tuottavuutta, koska siinä toimitaan tapahtumatasolla sen sijaan, että toimittaisiin signaali- tai nastatasolla. Abstraktiotasoa nostamalla saadaan myös nopeutta verifiointiin. Itsetarkastus (self-checking) ja kohdistettu satunnaistestaus voidaan saada aikaan helposti. [1.]

Koodin kattavuus

Koodin kattavuus (code coverage) -analyysi tarjoaa mahdollisuuden määrittää tietty toiminnallinen peitto, jonka tietty testisarja saa aikaan kun se asetetaan määrättyyn suunnitteluun. Analyysityökalun avulla saadaan arvo prosentuaalisesta peitosta jokaiselle arvioidulle ominaisuudelle (attribuutille) sekä lista suunnittelun testaamattomista tai osin testatuista alueista. Koodin kattavuus -analyysi tehdään suunnittelun RTL-tason (register-transfer level) näkymälle. Koodin kattavuus -analyysin avulla saadaan arviointi testisarjan laadulle sekä myös suunnittelun testaamattomat alueet. [1.]

Laitteiston ja ohjelmiston yhteisverifiointi

SoC-järjestelmän suunnittelussa luodaan abstrakti malli suunnittelusta ja se simuloidaan. Abstrakti toiminnallisuus liitetään sitten järjestelmän yksityiskohtaiseen arkkitehtuuriesitykseen. Seuraavaksi suunnittelu jakaantuu HW- ja SW-komponentteihin. Laitteistosuunnittelijat käyttävät VHDL- tai Verilog-ohjelmointia. Laitteistosimulaattoria käytetään verifioimiseen. Ohjelmistosuunnittelijat puolestaan koodaavat ohjelmamoduuleja esim. Assembly- tai C++-kielen avulla ja käyttävät prosessorimalleja tai ICE-menetelmää testatakseen suunnitteluaan. Tyypillisesti SW-suunnittelija odottaa sitten prototyypin valmistumista, jotta lopullinen järjestelmä voidaan integroida. [1.]

Integrointiprosessissa on kuitenkin usein ongelmia. Saattaa olla, että esimerkiksi spesifikaatioiden noudattamisessa on ollut väärinkäsityksiä tai suunnittelua on muutettu myöhäisessä vaiheessa. Korjaaminen tässä vaiheessa huomatussa virheessä vie aikaa ja esimerkiksi laitteistoon tehtävät muutokset ovat

usein kalliita. Laitteiston ja ohjelmiston yhteisverifiointi (HW/SW Co-verification) varhaisessa vaiheessa suunnittelua ehkäisee tällaisia ongelmatilanteita. Laitteiston ja ohjelmiston yhteisverifioinnissa integroidaan ja verifioidaan samanaikaisesti. Yhteisverifiointiympäristössä on käytössä graafinen käyttöliittymä, joka on yhdenmukainen yleisten laitteistosimulaattoreiden sekä ohjelmiston simulaattoreiden ja debuggereiden kanssa, joita suunnittelussa (sekä laitteiston että ohjelmiston kanssa) käytetään. [1.]

Emulointijärjestelmät

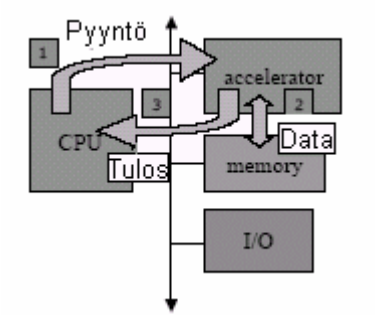
Emulointi- eli jäljittelyjärjestelmät ovat erikseen suunniteltuja laitteisto- ja ohjelmistojärjestelmiä. Ne sisältävät konfiguroitavaa logiikkaa, kuten ohjelmoitavia FPGA-piirejä. Ne matkivat kohteena olevan suunnittelun käyttäytymistä ja voivat jäljitellä sen toiminnallisuutta. Ne voidaan suoraan liittää suunnitteluympäristöön, jossa lopullinen suunnittelu toimii. Koska järjestelmä toteutetaan HW:llä, ovat suoritusnopeudet huomattavasti suurempia kuin SW-pohjaisilla simulaattoreilla. Emulointiympäristön on kuitenkin oltava riittävä, jotta emulointi voidaan toteuttaa maksiminopeudella. Emuloinnin avulla saattaa myös olla nopeudesta huolimatta vaikeampi löytää virheitä, sillä emulaattoreiden debuggaus-ympäristö ei välttämättä aina yllä ohjelmistopohjaisten simulaatioiden tasolle. [21.]

Pikamallinnusjärjestelmät

Pikamallinnusjärjestelmän (rapid prototyping system) avulla voidaan mallintaa tarkasti aiotun SoC:n prototyyppiä. Suunnitelman laitteistototeutus rakennetaan ja sitä käytetään ohjelmiston kehittämiseen ja debuggaukseen. Prototyypit on tapana rakentaa yhdestä tai useammasta FPGA-piiristä. Simulointinopeus on suurempi kuin ohjelmistopohjaisissa simulaattoreissa tai yhteisverifioinnissa. ASIC-suunnittelijat, joiden suunnitelmat ovat erityisen monimutkaisia, vähentävätkin huomattavasti riskiä käyttämällä FPGA-piirejä prototyypin tekemiseen. [1.]

Laitteistokiihdyttimet

Laitteistokiihdyttimet (hardware accelerators) nopeuttavat määrättyä simulointitapahtumaa. Niiden avulla voidaan suorittaa koko suunnittelua tai osaa siitä. Laitteistokiihdytin on komponentti, joka työskentelee prosessorin kanssa yhdessä ja suorittaa tehtäviä huomattavasti nopeammin kuin prosessori. Se esiintyy laitteena väylällä (kuva 7). [22.]



Kuva 7. Laitteistokiihdytin (hardware accelerator) toimii yhdessä prosessorin kanssa. [22]

Useimmiten teollisuudessa sovellusta käytetään siten, että testiohjelmaa suoritetaan ohjelmiston avulla ja itse verifioitavaa suunnitelmaa suoritetaan laitteistokiihdyttimellä [5, s. 231].

Analogisten ja sekasignaalien simulointi

Elektroniikassa, erityisesti kuluttajille suunnatussa (esim. ajoneuvojen järjestelmät), on käytössä analogia- ja sekasignaaliliityntöjä digitaalisten lohkojen lisäksi. Sulautetut AMS (analog/mixed signal) -lohkot SoC:ssa ovat haastavia simulaatiolle, sillä on verifioitava sekä digitaalisia että analogisia piirejä. [1.]

AMS-piirit ovat yhteydessä ulkopuoliseen maailmaan mittaamalla analogisia signaaleja ja muuttamalla ne digitaalisiksi sekä päinvastoin. Lohkoissa on esimerkiksi A/D- ja D/A-muuntimia ja PLL-komponentteja. Nykypäivän piiriteknologia sallii AMS-lohkojen sisällyttämisen SoC:iin yhdessä digitaalisten lohkojen kanssa.

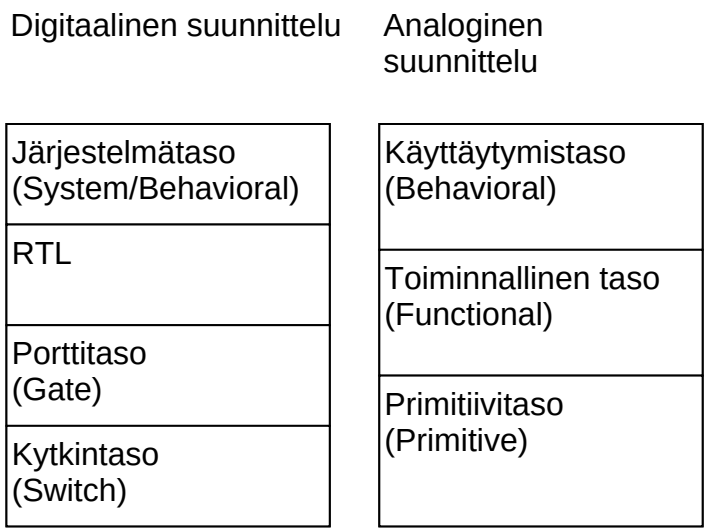
Analogisen simulaation tarkoitus on verifioida, että DUT (design under test) toimii tuotteen määrittelyjen mukaisesti (jännite, virta ja ajoitusspesifikaatiot). Digitaalisessa simuloinnissa simuloinnin tarkoitus on verifioida suunnittelun toiminnallisuus ennalta määrättyjen sisääntulovektoreiden ja ajoitusspesifikaation mukaan. Jännitteen tarkkuus ei ole mielenkiinnon kohteena. Sekasignaalien simulointiympäristön on toimittava analogisen ja digitaalisen simuloinnin sekä niiden vuorovaikutuksen parissa. Nykyisin monet sekasignaalien simulointivalinnat yhdistävät digitaaliseen simulaattoriin analogisen simulaattorin. Simulaattorit suorittavat erillistä prosessia ajon aikana ja pääprosessi kontrolloi niiden välistä datan siirtoa. [1.]

Digitaaliset ja analogiset suunnitelmat kuvataan eri top-down-abstraktiotasoina suunnittelua ja simulaatiota varten. Digitaalisen suunnittelun abstraktiotasot ovat järjestelmä- eli käyttäytymistaso, rekisterinsiirtotaso (RTL), porttitaso ja kytkintaso. [1.]

- **Järjestelmätaso** kuvaa suunnittelun käyttäytymisen muutamilla yksityiskohdilla rakenteellisessa toteutuksessa. Tätä tasoa käytetään järjestelmän tai suunnittelun perusperiaatteiden simulointiin ja toteamiseen sekä rakenteellisten määritysten luomiseen.
- **RTL-taso** kuvaa suunnittelun toiminnan rekistereiden, kombinaatiopiirien, väylien ja ohjauspiirien avulla puuttumatta porttitason toteutukseen. Simulaatio tehdään RTL-tasolla suunnittelun logiikan ja ajoituksen varmistamiseksi.
- **Porttitaso** kuvaa loogisten porttien rakenteellisen yhteyden avulla toimintaa, ajoitusta ja suunnittelun rakennetta. Loogiset lohkot toteuttavat Boolean-funktioita. Porttitasolla verifioidaan yksittäisten signaalipolkujen ajoitusta.
- **Kytkintaso** kuvaa loogisia piirejä toteuttavien transistoreiden yhteyksiä. Transistorit on mallinnettu on-off-piireinä. Tasoa käytetään kriittisten signaalipolkujen tarkemman ajoituksen verifiointia varten. [1.]

Analogisen suunnittelun abstraktiotasot ovat käyttäytymistaso, toiminnallinen taso ja primitiivitaso. **Käyttäytymistaso** on sisällöltään sama kuin digitaalisessakin suunnittelussa. **Toiminnallinen taso** kuvaa suunnittelun toiminnan ilman

transistoritason toteutuksen yksityiskohtia. Taso vastaa digitaalisen suunnittelun RTL-tasoa. **Primitiivitaso** kuvaa piirin toimintaa vastuksien virta-jännite-käyttäytymisen, kondensaattorien, kuristimien ja puolijohdekomponenttien sekä niiden yhteyksien suhteen. Kuvassa 8 on esitetty digitaalisen ja analogisen suunnittelun abstraktiotasot. [1.]



Kuva 8. Digitaalisen ja analogisen suunnittelun abstraktiotasot [1]

Tyypillisessä sekasignaalien simulointiympäristössä on useita elementtejä. Analoginen simulaattori simuloi analogisia piirejä, digitaalinen/looginen simulaattori simuloi logiikkaa. Edellisten kommunikaatiosta huolehtii linkkimenetelmä. Mukana ovat myös laite- ja liityntäelementtien mallikirjasto sekä simulaation testipenkki. Simulaation tulokset ja lähtöjen aaltomuodot saadaan lopuksi analysoitaviksi. [1.]

Valittaessa simulointiympäristöä sekasignaalien simulointiin on olemassa muuttujia, jotka vaikuttavat valintaan. Nopeus ja tarkkuus sekasignaaliympäristössä riippuvat käytettävästä analogiasimulaattorista. Suoritukseen vaikuttavat mm. suunnittelun kapasiteetti, ajoitusrajoitukset ja kellon nopeus. Suunnittelun vaatimien kriittisten kirjastomallien olemassaolo kannattaa tarkistaa, samoin näiden tarkkuus ja toiminta laitteiston kanssa. Suunnitteluympäristö käyttää liityntämalleja, jotka on lisätty suunnittelun analogisten ja digitaalisten osien väliin. Ne estävät ongelmia, jotka johtuvat eroista signaalin esitys-

muodossa. Digitaalinen simulaattorihan käyttää binääristä esitysmuotoa signaaleilla. Tämä poikkeaa täysin analogisen simulaation signaalin esityksestä. Digitaaliset signaalit voivat aiheuttaa epävakautta analogisessa simulaatiossa sillä niissä tapahtuu äkillisiä jännitteen muutoksia solmujen tilanvaihdoksien yhteydessä. Liityntämallin ominaisuuksiin on kiinnitettävä huomiota. Ne huolehtivat myös eri tilojen hallinnasta. Digitaalinen simulaattori käsittelee logiikkatasoja 0, 1 ja X (don´t care) sekä korkeaimpedanssista Z-tilaa. Analoginen simulaattori tarvitsee tarkat jännite- ja virtatasot kaikissa piirin solmukohdissa. X- ja Z-tilat aiheuttavat siis ongelmia ja sopivat jännitteet onkin pystyttävä tarjoamaan. [1.]

On myös tarkistettava simulaatioympäristön tukema ohjelmointikieli. Samaa kieltä voidaan käyttää sekä analogisen että digitaalisen suunnittelun kuvaukseen ja mallinnukseen. Esim. Verilog- ja VHDL-kielessä on AMS-laajennuksia (VHDL-AMS, standardi IEEE 1076.1 - 1999 ja Verilog-A/MS, standardi 1.3). Tärkeä ominaisuus simulointiympäristölle on myös tulosten analysointiominaisuudet sekä nopea virheiden löytyminen. Käyttäjän pitäisi pystyä näkemään sekä analogisten että digitaalisten signaalien aaltomuodot. Tällä hetkellä sekasignaalien simulointiin ei ole helppoa löytää käyttökelpoisia ja helposti omaksuttavia sovelluksia. [1.]

3.1.2 Staattiset tekniikat

Staattiset menetelmät verifioinnissa ovat tekniikoita, joiden kanssa ei käytetä testivektoreita. Ne perustuvat piirikuvauksen analysointiin. Staattisiin verifiointitekniikoihin kuuluvat lint-tarkastus (lint checking) sekä staattinen ajoitusverifiointi (static timing verification). [1.]

Lint-tarkastus

Lint-tarkastus (nimi tarkoittaa nukkaa, nöyhtää ja se on johdettu ajatuksesta, että ohjelmasta tulee puhtaampi, kun ylimääräiset haituvat poistetaan [23]) on ohjelmointityökalu, jonka avulla voidaan varmistaa, että käytetty koodi täyttää tarvittavat standardit ja ohjeet ja se voidaan tehdä jo suunnittelun varhaisessa

vaiheessa. Se löytää yksinkertaiset virheet suunnittelukoodista kuten alustamattomat muuttujat, porttien ristiriidat ym. ohjelmointivirheet. [1.]

Staattinen ajoituksen verifiointi

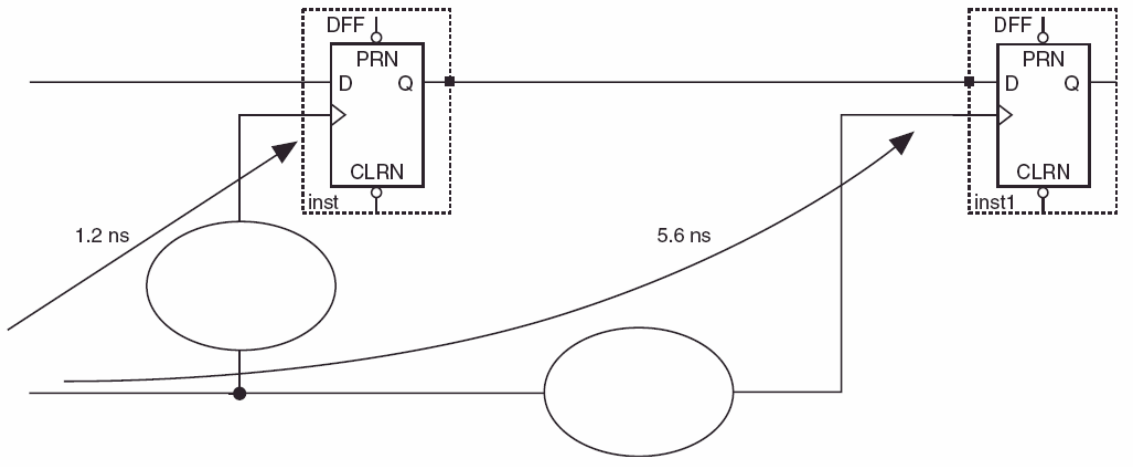
Kun on suoritettu toiminnallista simulointia suunnittelulle ja havaittu, että suunnitelma on sille annettujen toiminnallisten määrittelyjen mukainen, on aika suorittaa ajoitusverifiointia. Sen avulla nähdään täyttääkö suunnitelma ajoituksselliset vaatimukset. Selvitetään myös voidaanko viiveet optimoida siten, että saadaan piirille parempi suorituskyky. Tärkeitä menetelmiä ajoituksen verifiointiin ovat dynaaminen ajoitussimulaatio sekä staattinen ajoitusanalyysi (static timing analysis). [24.]

Dynaamista simulaatiota voidaan käyttää sekä asynkronisissa että synkronisissa suunnitteluissa. Se vaatii kattavat sisääntulovektorit tarkistaakseen ajoituspiirteet suunnitelman kriittisillä poluilla. Dynaamisen ajoitussimulaation avulla voidaan verifioida myös toiminnallisuutta. [24.]

Ennen staattisen ajoitusanalyysin yleistymistä varmennettiin piirin toiminta toivotulla nopeudella käyttämällä simulointia. Ongelmana oli simulaation hitaus porttitasolla sekä huono kattavuus. Simulointi ei myöskään voi todistaa, ettei virheitä löydy; se osoittaa vain, ettei ongelmia ole kun testaus tehdään määrätyllä tavalla. Staattinen ajoitusanalyysi kehitettiin vaihtoehdoksi porttitason simulaattorille ennen tuotteen valmistusta. Se tarkistaa jokaisen suunnitelman polun ajoitusrikkomusten varalta. Se ei kuitenkaan tarkista suunnitelman toiminnallisuutta. Se ei tarvitse testivektoreita. Toiminnallista simulointia ja ajoitusverifiointia voidaankin suorittaa rinnakkain. Tämä metodi on huomattavasti nopeampi kuin dynaaminen ajoitusanalyysi, koska testivektoreita ei tarvitse generoida ajoituksen verifiointia varten. Staattisen ajoitusanalyysin avulla voidaan kuitenkin suorittaa verifiointia ainoastaan synkronisten piirien ajoitusvaatimuksille. Asynkronisille suunnitteluille voidaan suorittaa rajoittunutta analyysiä, mutta toiminnallisuus ei ole tällöin taattu ja piirille onkin suoritettava lisäksi verifiointia simuloinnin avulla. [24, s. 2 – 9.]

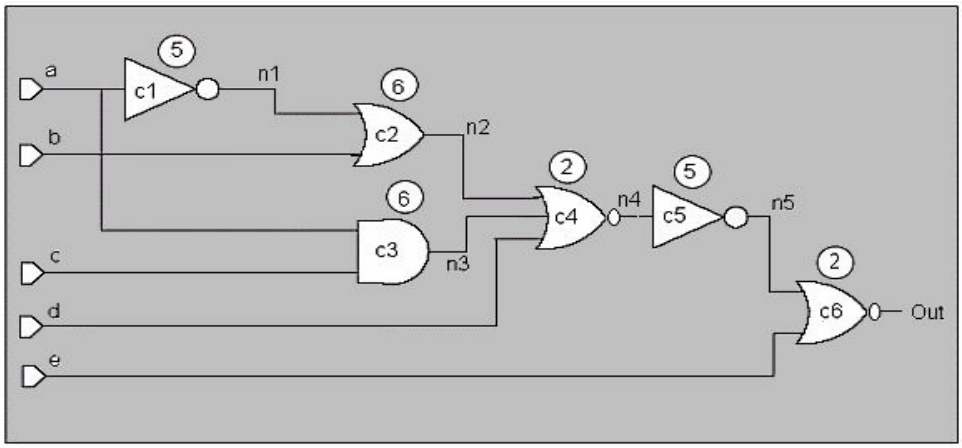
Staattinen ajoitusanalyysi verifioi viiveet piirin sisällä. Se käyttää kytkentälistaa ja kirjastomalleja sisääntuloinaan. Jokaisella suunnittelun tallennuselementillä ja lukkopiirillä on ajoitusvaatimuksia, kuten asetusaika, pitoaika ja eri viive-ajoitukset. Staattisen ajoitusanalyysin avulla löydetään häiriöt kellosta, rikkomukset asettumis- ja pitoajoissa, hitaat polut ja kellon vääristymät (skew). [24, s. 2 – 9.]

Kellon vääristyminen (skew) on eroavuus kellosignaalin saapumisajoissa kahdelle eri rekisterille. Näin tapahtuu kun kahdella kellosignaalipolulla on eri pituudet (kuva 9). Tapahtuma on yleinen suunnitteluissa, joissa on kellosignaaleja, joita ei ole reititetty globaalisti. [25.]



Kuva 9. Erilaiset kellosignaalin saapumisajat (1,2 ns ja 5,6 ns) [25]

Staattisen ajoitusanalyysin perustavoite on mitata jokaisen polun viive. Arvo porttiviiveille saadaan toimittajan kirjastosta. Yhteyksien viiveet joko arvioidaan synteesin aikana tai saadaan sijoituksen ja reitityksen jälkeen. Kuvassa 10 on esitetty mitatut porttiviiveet asynkronisesta logiikasta.



Kuva 10. Staattinen ajoitusanalyysi mittaa porttiviiveet piirin läpi. [24, s. 7]

Kuvassa 10 on kuvattu useiden sisääntulojen ja ulostulon muodostamat eri polut. Etenemisviive portin läpi on kuvattu ympyrän sisään portin viereen. Jokaisella kytkentäosalla (net) on yhden aikayksikön pituinen viive. Voidaan nähdä, että maksimiviive syntyy polulla a – Out, kun polku kulkee porttien c1-c2-c4-c5-c6 kautta. Viive on 26 aikayksikön pituinen. Minimiviive syntyy tulosta e - Out, jolloin kuluu neljä aikayksikköä. Analyysin jälkeen analysaattori vertaa aikoja suunnittelijan määrittelemiin oikeanlaisen toiminnan vähimmäis- ja enimmäisaikoihin. Jos jokin polku ei vastaa määrittelyjä, siitä raportoidaan ongelmana. [24.]

Jos kysymyksessä on synkroninen piiri, lasketaan vähimmäis- ja enimmäisajat samalla tavoin. Lisäksi tulevat kuitenkin kiikkujen viiveet ja asettumisajat. Suunnittelijan määrittelemä kellojakso asettaa ajan, jona aikana edetään rekistereiden välillä. Kellon aktiivisella reunalla data etenee kiikkujen porttien ja yhteyksien läpi kohti seuraavaa kiikkua. Datan on saavuttava ajoissa, sillä valmistelutulojen on oltava vakioita asettumisajan verran ennen kellotulon muutosta. Jos kellojakso on 10 ns ja data kulkee kahden kiikun väliä 15 ns ajan, on ilmeistä, että data on myöhässä. Vaikka data saapuisikin seuraavalle kiikulle 10 ns ajassa, on se silti myöhässä, sillä se on rikkonut kiikun asettumisaikaa. [24, s. 9.]

Staattinen ajoitusanalyysi suoriutuu verifiointitehtävästään merkittävästi nopeammin kuin porttitason simulaatio. Vaikka se rajoittuu vain synkronisille piireille,

on siitä tullut yleinen työkalu monien valmistajien ohjelmistoissa. Esimerkkejä EDA (Electronic Design Automation) -työkaluista ovat Synopsys PrimeTime [26], joka on erillinen staattinen ajoitusanalysaattori ja Cadense Design Systems Pearl [20].

3.1.3 Formaalit tekniikat

Suunnittelun verifioinnissa on toisinaan vaikea löytää ohjelmointivirheitä, jotka riippuvat erityisestä järjestyksestä tapahtumissa. Tällaisilla näkymättömillä ja vaikeasti selvitettävillä virheillä voi olla voimakas vaikutus jos niitä ei löydetä tarpeeksi varhaisessa vaiheessa. Formaalit tekniikat auttavat tällaisten virheiden löytymisessä. Formaaleja verifiointimenetelmiä ovat väittämien osoitustekniikka (theorem proving technique), formaali mallitarkastus (formal model checking) ja formaali vastaavuustarkistus (formal equivalence checking). [1.]

Teoreeman osoittamistekniikkaa tutkitaan vielä. Tekniikkaa osoittaa että suunnittelu täyttää funktionaaliset vaatimukset sallimalla käyttäjän rakentaa kokeilu suunnittelun käyttäytymisestä käyttämällä teoreemia, väittämiä. [1.]

Formaali mallitarkastus hyödyntää formaaleja matemaattisia tekniikoita verifioidakseen suunnittelun ominaisuuksia. Mallin tarkastustyökalu vertaa suunnittelun käyttäytymistä suunnittelijan määrittelemiin loogisten ominaisuuksien sarjaan. Määritellyt ominaisuudet on poimittu suunnittelun spesifikaatiosta. Formaali mallitarkastus sopii hyvin monimutkaisiin ohjausrakenteisiin, kuten esimerkiksi dekoodereille ja prosessorin ja oheislaitteiden välisille silloille. Tarkastustyökalut eivät vaadi testipenkkiä tai vektoreita. Verifioitavat ominaisuudet määritellään kysymysten muodossa. Kun virhe löytyy, generoidaan täydellinen jäljitys alkutilasta aina siihen tilaan asti, jolloin määritellyssä ominaisuudessa tapahtui virhe. Formaali mallitarkastus ei poista simuloinnin tarvetta, mutta se on täydentävä osa. Siinä ei ole ajoitukseen liittyvää informaatiota. [1.]

Formaali vastaavuustarkistus testaa vastaavuutta saman logiikkasuunnittelun kahden erilaisen näkymän välillä. Se käyttää matemaattisia malleja vertailu- ja muunnellun suunnittelun vastaavuutta tutkiessaan. Voidaan tutkia vastaavuutta

eri toteutuksien välillä (RTL - RTL, RTL - portti ja portti - portti). Koska työkalu vertaa nimenomaan vertailu- ja muutettua suunnittelua, on tärkeätä, että vertailusuunnittelu on toiminnallisesti oikea. Tekniikka ei tarvitse testipenkkiä tai vektoreita ja se löytää teoriassa kaikki virheet. Se on myös nopea verrattuna simulaatioon. Ajoitustietoja se ei kuitenkaan tarjoa. Vastaavuustarkistus on eniten käytetty formaali tekniikka. Formaalin verifioinnin EDA-työkaluja ovat esimerkiksi Synopsis Formality [26] ja Cadence Design Systems FormalCheck [20]. [1.]

3.1.4 Fyysinen verifikaatio ja analyysi

Fyysisessä verifioinnissa analysoidaan ajoitusta, signaalin laatua, ylikuulumista, jännitteen tipahtamista virtalähteen linjan resistanssin vuoksi, tehoa ja muita vastaavia seikkoja. Löytyneet virheet korjataan. Esimerkiksi sisäisten yhteyksien loisvirtojen vaikutukset on korjattava, sillä yhteysviiveet hallitsevat porttiviiveitä. [1.]

3.1.5 Verifiointivaihtoehtojen vertailu

Eri verifiointitekniikat tarjoavat erilaisia etuja ja ominaisuuksia. Taulukossa 1 on esitetty vertailua eri verifiointimenetelmien välillä. [1.]

Taulukko 1. Verifiointivaihtoehtojen vertailua [1]

	Event-based Simulation	Cycle-based Simulation	Hardware Accelerators	Emulation	Formal verification	Static timing verification
Function	Yes	Yes	Yes	Yes	No	No
Abstraction Level	Behavioral, RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	Gate
Functional Equivalence	Yes	Yes	Yes	Yes	Yes	No
Timing	Yes	No	Yes/No	No	No	Yes

Gate Capacity	Low	Medium	High	Very high	High	Medium
Run Time	<10 Cycles	1K Cycles	1K Cycles	1M Cycles	Medium	High
Cost	Low	Medium	Medium	High	Medium	Low

Tapahtumapohjainen simulaatio sopii parhaiten asynkronisille suunnitteluille. Se keskittyy sekä suunnittelun toimivuuteen että ajoitukseen. Pienille suunnitteluille se on nopein tekniikka. Jaksopohjainen simulaatio keskittyy puolestaan ainoastaan suunnittelun toiminnan varmentamiseen eikä ajoitusverifiointia ole. Keskitason kokoisille suunnitteluille menetelmä on sopiva. Formaali verifiointi ei vaadi testipenkkejä eikä testivektoreita. Se sisältää mallin tarkistuksen ja vastaavuustarkistuksen. Mallitarkastus verifioi ohjauslogiikkaa ja se vaatii, että suunnittelun ominaisuudet ja rajoitukset on määritelty. Nykyiset mallintarkastustyökalut sopivat pienille suunnitteluille, mutta vastaavuuden tarkistus sopii suurempiin suunnitteluihin vertaamaan kahta eri versiota. Ajoitusinformaatiota ei menetelmässä ole. [1.]

Emulaation avulla suoritettu verifiointi on simulaationopeudeltaan huomattavasti suurempi kuin muilla tekniikoilla. Se voi käsitellä kapasiteetiltaan suuria suunnitteluja ja testivektoreita. Ajoitustietoa ei ole. Nopean prototyypin menetelmää voidaan käyttää ohjelmiston kehityksessä. Sen avulla päästään SoC-alustaan käsiksi jo aikaisessa vaiheessa. Ajoitus on tälläkin menetelmällä tutkittava erikseen. [1.]

3.2 FPGA-perustaisen SoC:n verifiointi implementoinnin jälkeen

Vaikka suunnitelmaa olisi simuloitu kuinka paljon tahansa, on korttitasolla (in-system) tapahtuva testaus välttämätöntä. Jokaisella suunnitelmalla on omat ajoitusmuunnelmansa ja sähköinen ympäristönsä. [14.]

ASIC- ja FPGA-suunnittelussa on usein eroja suunnittelun implementoinnin jälkeisessä verifioinnissa. ASIC-suunnittelun verifiointiin ei aina sisällytetä laitteen korttitason testausta ja sen kaltaista testausta tapahtuukin usein kuukausia sen jälkeen kun suunnitelman toiminta on varmistettu simulaation avulla. Suunnitte-

lun ja korttitason testauksen välillä on merkittävä viive. Korttitason testaus aloitetaan kun osa on valmistettu. [14.]

FPGA:n suunnittelussa on in-system-testaus korttitasolla haastava osa verifiointiprosessia. Varsinaisen järjestelmän ajo sen todellisella nopeudella on täysin eri tilanne verrattuna testausympäristössä toteutettuun simulaatioon. Eräs suuri FPGA-piirin etu ASIC-piireihin nähden on sen uudelleenohjelmoitavuus. Ominaisuus vähentää myös simuloinnin kuormitusta suunnittelussa. Suunnittelun kohteena olevaa korttia voidaankin pitää simulaation kiihdyttimenä. Korttitasolla päästään nopeuteen, joka on suunnittelun oma kellotaajuus; tehokkaatkaan simulaattorit eivät yllä yhtä suuriin nopeuksiin. Ominaisuus onkin aiemmin johtanut ”tilanteen mukaan” (ad-hoc) -lähestymistapaan simuloinnissa. Nykyisten miljoonien porttien suunnittelujen kanssa ei lähestymistapa enää toimi. Simulaatio onkin tarpeellista ennen kuin suunnitelma toteutetaan laitetasolla. [14.]

Sulautetut prosessorit tarjoavat suunnittelijalle lisääntyvää toiminnallisuutta, mutta FPGA:lle sisäänrakennettuna ne aiheuttavat erikoisia ongelmia. Kun suunnitelmaa tehdään piiritason sulautetun prosessorin kanssa, käytetään usein ICE-pohjaista (In Circuit Emulation) debuggausympäristöä. Usein vaaditaan pääsyä piirin pinneihin tai JTAG-porttiin. Kun prosessorin ydin on sisäänrakennettu FPGA:lle, ei välttämättä olekaan tarpeellista liityntää ICE-työkalua varten. Onkin löydettävä uusia lähestymistapoja. [14.] [27.]

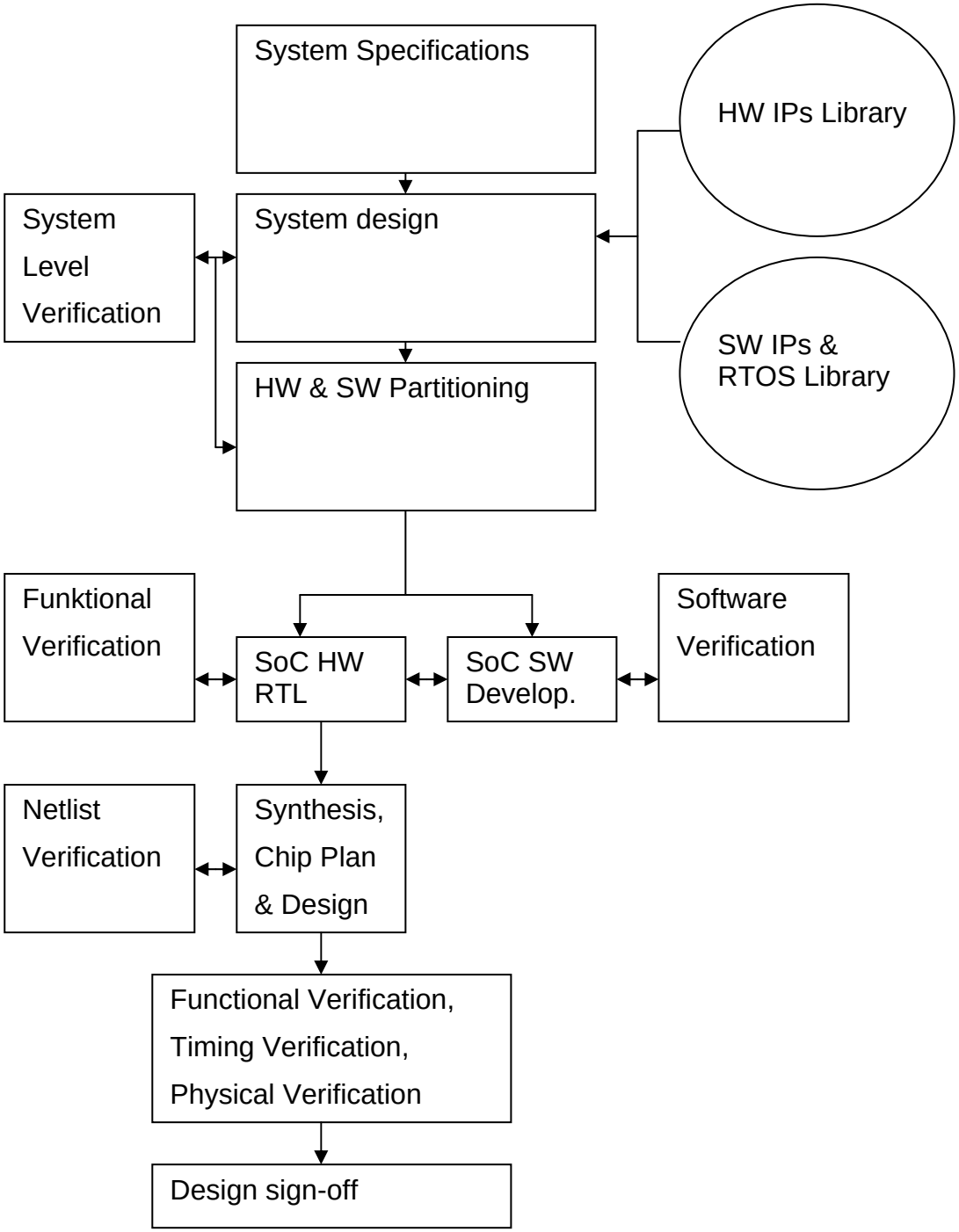
Laitteiston verifiointityökalujen avulla suoritetaan järjestelmän debuggausta. Tarvitaan tietoa siitä, mitä FPGA:n sisällä tapahtuu, ja mahdolliset virheet on saatava korjattua. Tietoa kerätään mittaamalla logiikkaa ja varastoimalla tulokset. Eräs tapa sisäisten signaalien tutkimiseen on logiikka-analysaattori. Sen avulla analysointi tapahtuu tuomalla signaalit laitteen pinneihin. Logiikka-analysaattorin avulla voidaan sitten kerätä tietoa käyttämättä tutkittavan kohteen logiikkaresursseja. Signaalit on kuitenkin kytkettävä manuaalisesti. Alemmalla hierarkiatasolla olevat solmukohdat on reititettävä ylemmälle tasolle muokkaamalla suunnitelman pinnejä. Analysaattorin käyttöä rajoittaa myös vapaiden nastojen määrä laitteessa sekä kortille sijoitettujen nastojen määrä. Signaalien nimet on merkittävä, jotta voidaan jäljittää, mikä solmukohta on nä-

kyvissä milläkin rivillä analysaattorin esittämässä kuviossa. Jos mittapaikkoja muutetaan, on koko prosessi tehtävä uudelleen. [27.]

Muutamat ohjelmoitavan logiikan valmistajat tarjoavat työkaluja laitteiston debuggaukseen käyttämällä ohjelmointiporttia tietojen keräämiseen. Ohjelmointiportin kautta liitytään sisäisiin solmukohtiin, eikä ulkoisia pinnejä tarvita. Suunnitelman solmukohtien tulokset siirretään JTAG-porttiin. Työkalut tukevat useiden kellojen taajuuksien näytteistystä ja useampia liipaisutasoja kuin tavallinen logiikka-analysaattori. Ne kuitenkin käyttävät jonkin verran tutkittavan laitteen logiikkaresursseja, jotka eivät siten ole käytettävissä muuhun suunnitteluun. [27.]

3.3 Verifiointimenetelmät

Järjestelmälle tehdyt määrittelyt ohjaavat verifiointistrategiaa. Verifiointisuunnitelma olisikin hyvä aloittaa rinnakkain määrittelyjen luomisen kanssa. Soc-piirin verifiointimenetelmät on esitetty kuvassa 11.



Kuva 11. SoC:n verifiointimenetelmät [1]

Järjestelmän suunnittelussa järjestelmän käyttäytyminen mallinnetaan määrittelyjen mukaisesti. Verifiointi suoritetaan käyttäytymistä simuloivan testipenkin avulla. Testipenkki on luotu esimerkiksi HDL-, C++- ja Vera-kielillä. Verifioinnin jälkeen järjestelmä liitetään sopivaan arkkitehtuuriin käyttämällä

esimerkiksi kirjastoissa saatavilla olevia IP-lohkoja (HW- tai SW-lohkoja). Seuraavaksi laitteisto- ja ohjelmistosuunnittelu erotetaan. [1.]

Laitteiston verifioinnissa RTL-koodi ja testipenkki saadaan järjestelmän suunnittelusta. Testipenkki muunnetaan sopivaan formaattiin, jotta se pystyy verifioimaan RTL-koodia ja suunnitelman toiminnallisuus verifioidaan. RTL-koodin verifiointi voi sisältää esimerkiksi lint-tarkastuksen, formaalin mallintarkastuksen ja loogisen simulaation. [1.]

Ohjelmiston verifioinnissa tarvittava ohjelmisto ja testaustiedostot saadaan ohjelmistosuunnittelijoilta. Verifiointi suoritetaan käyttäen pohjana järjestelmän määrittelyjä. Tässä vaiheessa on myös mahdollista suorittaa verifiointia ja laitteiston ja ohjelmiston yhteensovittamista. Menetelminä voidaan käyttää laitteiston ja ohjelmiston yhteisverifiointia, emulointia tai pikamallinnusta. [1.]

Laitteiston RTL-tason jälkeisen synteesin jälkeen luodaan porttitasoinen kytkentälista. Kytkentälista voidaan verifioida formaalin vastaavuustarkastuksen avulla käyttämällä RTL-koodia vertailusuunnitelmana, johon porttitason kytkentälistaa verrataan. Näin voidaan varmistaa eri tasojen looginen vastaavuus. Ennen suunnitelman lopullista valmistumista suoritetaan vielä toiminnallista ja ajoituksellista verifiointia sekä fyysistä verifiointia. [1.]

3.4 Lähestymistapoja verifiointiin

SoC-suunnittelussa käytetään useita lähestymistapoja verifiointiin. Näitä ovat järjestelmästä yksittäisiin komponentteihin etenevä suunnittelu (top-down) ja verifiointi sekä yksittäisistä komponenteista järjestelmään etenevä (bottom-up) verifiointi. SoC-suunnittelu on usein yhdistelmä molemmista. Muita lähestymistapoja ovat alustapohjainen (platform-based) verifiointi sekä järjestelmän rajapintapohjainen verifiointi. [1.]

Järjestelmästä yksittäisiin komponentteihin etenevä suunnittelu ja verifiointi

Lähtökohtana top-down-suunnittelussa on aina toiminnallinen määrittely (tyypillisesti paperipohjainen dokumentti). Määrittelyn pohjalta kehitetään yksityiskohtainen verifiointisuunnitelma. Järjestelmätason mallin toiminnallisuus verifioidaan systeemitason testipenkin avulla. Sen jälkeen suunnittelu voi hajaantua useiden abstraktiotasojen läpi, kunnes yksityiskohtainen suunnitelma on täydellinen. Mahdollisia abstraktitasoja voivat olla arkkitehtuurinen, HDL- ja kytkentälistamallit. Ylemmillä abstraktiotasoilla suunnittelu verifioidaan käyttämällä järjestelmän testipenkkiä, jota on laajennettu joka abstraktiotasolla niin, että se kykenee testaamaan kulloisenkin tason lisääntynyttä toimivuutta ja ajallisia erityisseikkoja. [1.]

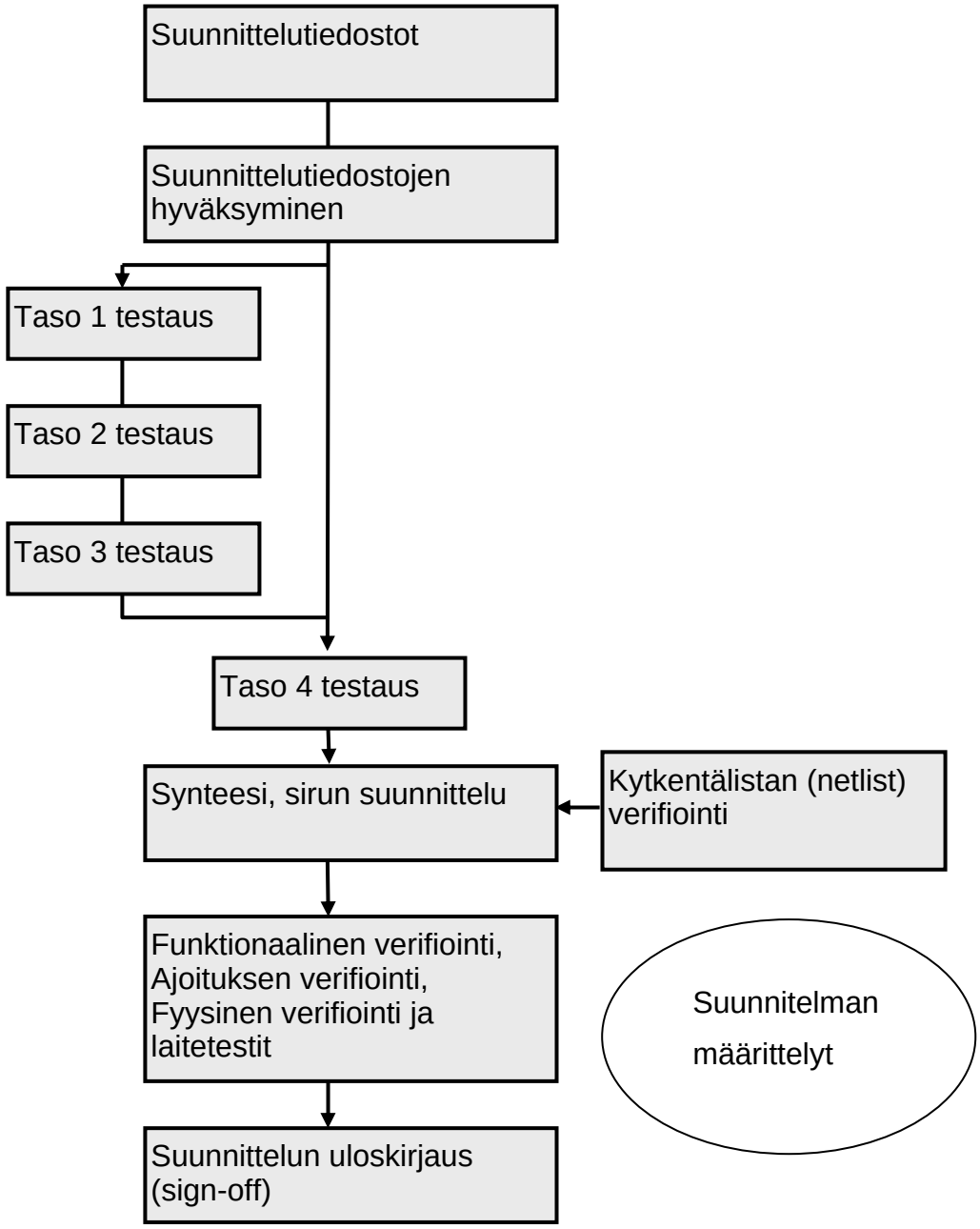
Tapahtumapohjaista verifiointia voidaan käyttää järjestelmätason sekä yhteyksien verifiointiin. Kun suunnittelu on hiottu HDL-tasolle ja halutaan testata seikkoja, joita ei simulaation avulla voida testata, voidaan lisätä täydentäviä verifiointitekniikoita. Näitä ovat esimerkiksi lint-tarkastus tai formaali mallintarkastus. Kun suunnittelun RTL on verifioitu, voidaan implementointinäkymiä verifioida formaalin vastaavuuden tarkastuksen tai simulaation avulla. Oikean implementaation varmistamiseksi tehdään vielä lopuksi ajoitus- ja fyysistä verifiointia sekä laitetestejä. [1.]

Suurentuneiden suunnitteluiden takia ei aina ole toteuttamiskelpoista suorittaa täyttä järjestelmätason testisarjaa yksityiskohtaisella mallilla. Tällaisissa tapauksissa voidaan simulaatioympäristöä jouduttaa esimerkiksi emulaation, laitteistokiihdyttimen tai nopean prototyypin avulla. Suunnittelu voidaan myös osittaa useisiin toiminnallisiin lohkoihin. Tällöin järjestelmäsimulaatiota voidaan käyttää sellaisessa tilassa, että vain yksi lohko kerrallaan tutkitaan yksityiskohtaisesti toisten lohkojen ollessa toiminnassa taas suunnittelun abstraktimallitasoilla. [1.]

Yksittäisistä komponenteista järjestelmään etenevä verifiointi

Yksittäisistä komponenteista järjestelmään etenevä (bottom-up) verifiointi on yleisesti käytössä. SOC-suunnittelut on rakennettu monista yksiköistä, lohkoista. Suunnittelut alkavatkin joukosta komponentteja, jotka on valittu määrättyjä paikallisia toimintoja varten [27]. Yksittäisten lohkojen toimintavaatimukset on johdettu koko piirille tehdystä määrittelystä. Lohkot suunnitellaan ja

verifioidaan niille asetettujen vaatimusten mukaisesti. Lopuksi koko piiri lohkoineen verifioidaan alkuperäiseen määrittelyn suhteen. Kuvassa 12 on esitetty verifiointilähestymistapa käyttäen yksittäisistä komponenteista järjestelmään etenevää tapaa. [1.]



Kuva 12. Yksittäisistä komponenteista järjestelmään etenevä (bottom-up) verifiointi [1, s. 27]

Ensimmäiseksi varmistetaan suunnittelutiedostojen yhteensopivuus ja soveltuminen suunnitelmaan. Seuraavaksi verifioidaan suunnittelutiedostot koodin lauseopillisten rikkomusten varalta (alustamattomat muuttujat, porttien ristiriidat

jne.). Seuraava askel riippuu suunnittelun abstraktiotasosta. Jos suunnittelu on yksityiskohtaisella tasolla, on sen läpäistävä neljä testitasoa. Korkeammalla abstraktiotasolla olevan suunnittelu voidaan heti viedä neljännelle järjestelmätason testaustasolle. [1, s. 26 - 28.]

Alin testaustaso verifioi erillään yksittäiset komponentit, yksiköt tai lohkot. Tarkoitus on tyhjentävästi testata komponentit kiinnittämättä huomiota niiden tulevaan integrointiympäristöön. Menetelminä käytetään determinististä simulaatiota, kohdistettua sattumanvaraista simulaatiota sekä lint- ja formaalista tarkastusta. Jos testit testipenkkeineen ovat IP-valmistajan tukemia, voivat ne sisältää koodin kattavuusanalyysin suunnittelun laadun varmistamiseksi. [1, s. 26 - 28.]

Toinen testitaso verifioi järjestelmän muistin kartoituksen ja sisäiset kytkennät. Testit voidaan suorittaa joko manuaalisesti tai generoida automaattisesti työkalulla, joka lukee muistikartan ja järjestelmätason yhteydet. Testien avulla tarkastetaan, että suunnittelussa oleviin rekistereihin voidaan kirjoittaa ja että niistä voidaan lukea piirillä olevan prosessorin avulla. Lisäksi kaikki kytkennät varmistetaan suorittamalla kirjoitus- ja lukutoimintoja suunnitelman tiedonvälityspoluilla. Kolmas taso varmistaa järjestelmätason suunnittelun perustoiminnallisuuden ja ulkoiset yhteydet. Testataan toiminnalliset pääpolut jokaisen toiminnallisen lohkon keskellä sekä kaikki I/O-nastat. [1, s. 26 - 28.]

Neljäs testauksen taso verifioi suunnittelun järjestelmätasolla. Tarkoituksena on tyhjentävästi testata integroidun suunnitelman toiminnallisuus. Jotta varmistettaisiin kaikenkattava testi, olisi erityisesti kiinnitettävä huomiota ”corner case” -tapauksiin (tilanne tai ongelma, joka esiintyy vain normaalien toimintamuuttujien ulkopuolella; erityisesti sellainen, jotka esiintyy, kun useat ympäristömuuttujat tai olosuhteet ovat yhtäaikaisesti ääritasoilla. [23]), suunnittelun epäjatkuvuuksiin, virhetiloihin, poikkeuskäsittelyihin ja ääriehtoihin. Testitasojen jälkeen oikean implementaation varmistamiseksi tehdään vielä kytkentälistan verifiointi, ajoitus- ja fyysistä verifiointia sekä laitetestejä. [1, s. 26 - 28.]

Alustapohjainen lähestymistapa

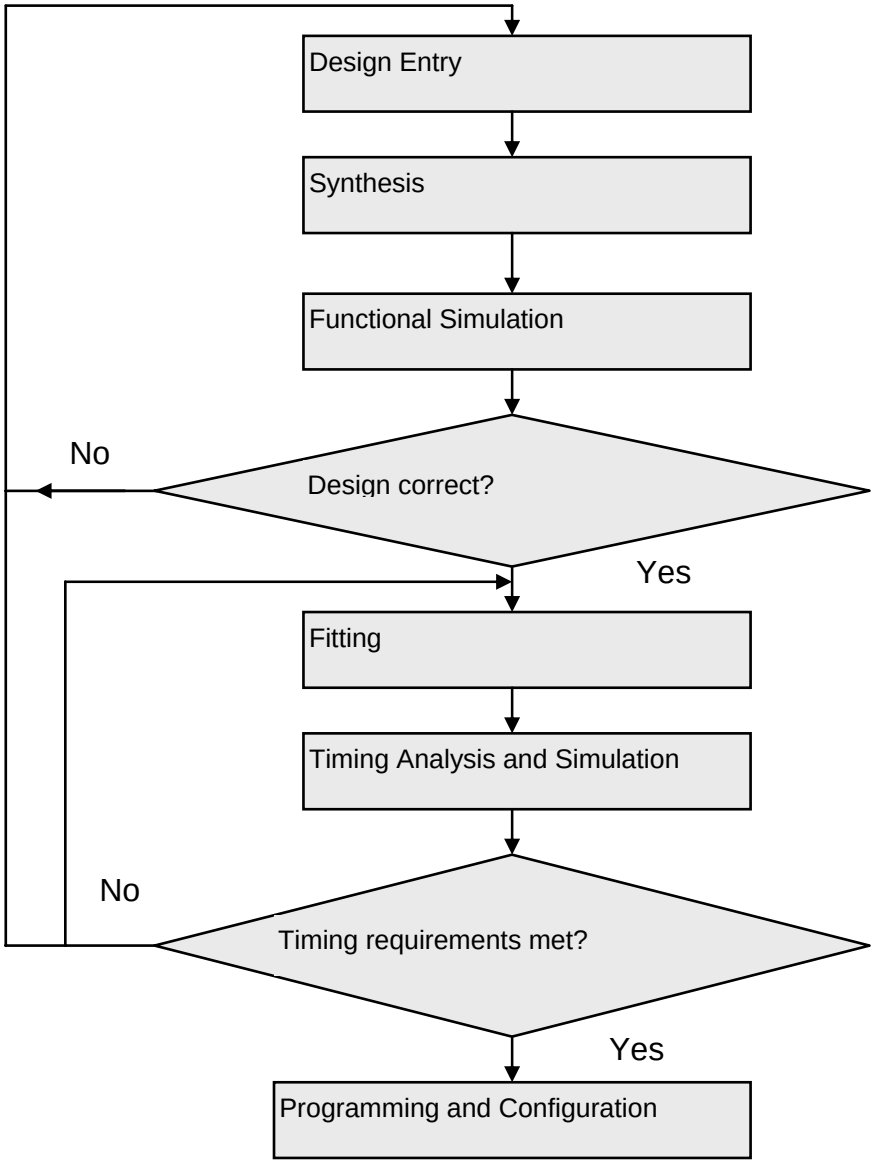
Alustapohjainen (platform-based) lähestymistapa on sopiva menetelmä, kun halutaan verifioida suunnitteluja, jotka perustuvat jo aikaisemmin verifioidulle alustalle. Myös käytettävät laitteiston ja ohjelmiston IP-lohkot on aiemmin verifioitu. Uudet IP:t lisätään mukaan ja verifiointi suoritetaankin perusalustan ja lisättyjen IP-lohkojen välisille yhteyksille. Perusalusta on voitu aiemmin verifioida käyttäen joko top-down-menetelmää tai bottom-up-menetelmää. [1.]

Järjestelmän rajapintoihin perustuvassa menetelmässä suunnittelussa käytetyt lohkot on mallinnettu liitäntätasollaan. Näitä malleja voidaan käyttää suunnitellun lohkon ja järjestelmän välisen liitynnän verifiointiin. Täten voidaan välttää ongelmia lopullisessa integroinnissa ja suunnitteluvirheet löydetään aikaisemmassa vaiheessa. [1, s. 28.]

4 ALTERAN STRATIX II:N VERIFIOINTI

4.1 Quartus II -ohjelmiston verifiointityökalut

Quartus II -ohjelmisto on Alteran oma suunnitteluohjelmisto sen omien piirien suunnitteluun. Sen avulla voidaan toteuttaa tietokonepohjaista (Computer Aided Design, CAD) suunnittelua. Kuvassa 13 on esitetty tietokonepohjaisella suunnittelulla toteutettu tyypillinen FPGA:n suunnitteluvuo. Suunnitteluvuo alkaa suunnittelun määrittelyllä, jonka jälkeen voidaan siirtyä piirisuunnitteluun. [28.]



Kuva 13. CAD-suunnittelun avulla toteutettu tyypillinen FPGA:n suunnitteluvuo. Suunnittelun määrittelyt toteutetaan ennen siirtymistä piirisuunnitteluun. [28]

Suunnittelutiedostot voidaan toteuttaa lohkokaavioesityksenä tai HDL-kielen avulla, kuten esim. VHDL ja Verilog. Synteesissä ohjelmisto selvittää tarvittavat logiikkaelementit (LE) ja niiden väliset yhteydet. Synteesin jälkeen voidaan toteuttaa toiminnallinen simulaatio (kunhan on generoitu tarvittava kytkentälista). Syntetisoitu piiri testataan simulaation avulla, jotta voidaan verifioida sen toiminnallinen oikeellisuus. Jos suunnittelussa havaitaan virheitä, voidaan palata tekemään korjauksia suunnittelutiedostoon ja tehdä uusi synteesi ja simulaatio. [28.]

Sijoitus- ja reititystoiminnon (Fitting) avulla sovitetaan suunnitelma varsinaiselle valitulle piirille. Toiminto valitsee tarvittavien loogisten elementtien paikat sekä elementtien väliset kytkennät. Ajoitussimulaatiota ja ajoitusanalyysiä voidaan suorittaa sijoitus- ja reititystoiminnon jälkeen. Ajoitusanalyysi analysoi viiveet eri signaalipoluilla ja sen avulla saadaan osoitus piirin suorituskyvystä. Ajoitusanalyysin avulla piiri testataan, jotta voidaan verifioida sekä sen toiminnallista virheettömyyttä että ajoitusta. Tässä vaiheessa voidaan asettaa ja muuttaa rajoituksia ja asetuksia sekä tarvittaessa palata suunnittelun aikaisempiin vaiheisiin, synteesiin tai sijoitus- ja reititystoimintoon. Kun suunnittelu on valmis, siitä voidaan tehdä ohjelmointitiedosto ja viimein ohjelmoida haluttu laite. [28.]

Quartus II -ohjelmiston kääntäjä (Compiler) sisältää toisistaan riippumattomia moduuleja, joiden avulla tarkistetaan suunnitelman virheet, tehdään syntetisointi logiikalle, sijoitetaan ja reititetään suunnitelma määrätylle Alteran piirille ja generoidaan tiedostot simulaatiota, ajoitusanalyysiä ja piirin ohjelmointia varten. Peruskääntäjään sisältyvät moduulit ovat Analysis & Synthesis, Fitter, Assembler ja Timing Analyzer. Niitä voidaan käyttää yhdessä tai erikseen. Jos suoritetaan täydellinen käännöstapahtuma, analyysi- ja synteesimoduuli (*Analysis & Synthesis*) tuottaa ensin informaatiota, joka määrittelee projektin suunnittelutiedostojen hierarkkiset yhteydet ja tarkistaa suunnitelmassa mahdollisesti esiintyvät perusvirheet. Seuraavaksi se luo rakenteellisen kartan suunnitelmasta ja yhdistää suunnittelutiedostot ”litistettyyn” tietokantaan prosessin nopeuttamiseksi. Sijoitus- ja reititystoiminto (*Fitter)* valitsee optimit polut, nastojen ja loogisten solujen sijoitukset ja reititykset määrätylle Alteran piirille. Seuraavaksi kääntäjä (*Assembler*) kääntää saadut sijoitukset ohjelmoinnissa

tarvittavaksi esitykseksi (image). Lopuksi suoritetaan ajoitusanalyysi (*Timing Analyzer*). EDA-kytkentälistat voidaan halutessa myös luoda käännöksen yhteydessä, jolloin generoidaan EDA-työkalujen (esim. ModelSim-Altera) tarvitsemia tiedostomuotoja. [29.]

Quartus II -ohjelmisto osaa generoida tunnettujen simulointi- ja ajoitusanalyysityökalujen valmistajien käyttämiä tiedostomuotoja: EDIF Input Files (.edf), Verilog Quartus Mapping Files (.vqm), VHDL Design Files (.vhd) ja Verilog Design Files (.v). Ohjelmistokehittäjien työkaluja ovat mm. Cadence Verilog-XL, Incisive NC-Verilog ja Incisive NC-VHDL, Mentor Graphics Tau, Mentor Graphics ModelSim, Innoveda SpeedWave, ja Synopsys VSS, VCS, VCS-MX, sekä PrimeTime. [29.]

Quartus II -ohjelmiston avulla voidaan suorittaa erilaista verifiointia suunnitelmalle sen eri vaiheissa sekä ohjelmiston omilla että muiden valmistajien työkaluilla. Verifiointimenetelmiä ovat simulaatio, ajoitusanalyysi, tehon arviointi ja analyysi sekä formaali verifiointi. Kun ohjelmointi on suoritettu piirille, voidaan suorittaa in-system -pohjaista virheiden hakua ja korjausta, debuggausta, eri työkaluilla. [29.]

Altera toimii yhteistyössä eri verifiointityökalujen valmistajien kanssa. Simulointiin, ajoitusanalyysiin, formaaliin verifiointiin ja in-system debuggaukseen on käytettävissä useita eri vaihtoehtoja kolmansien osapuolten työkalujen avulla. [13.]

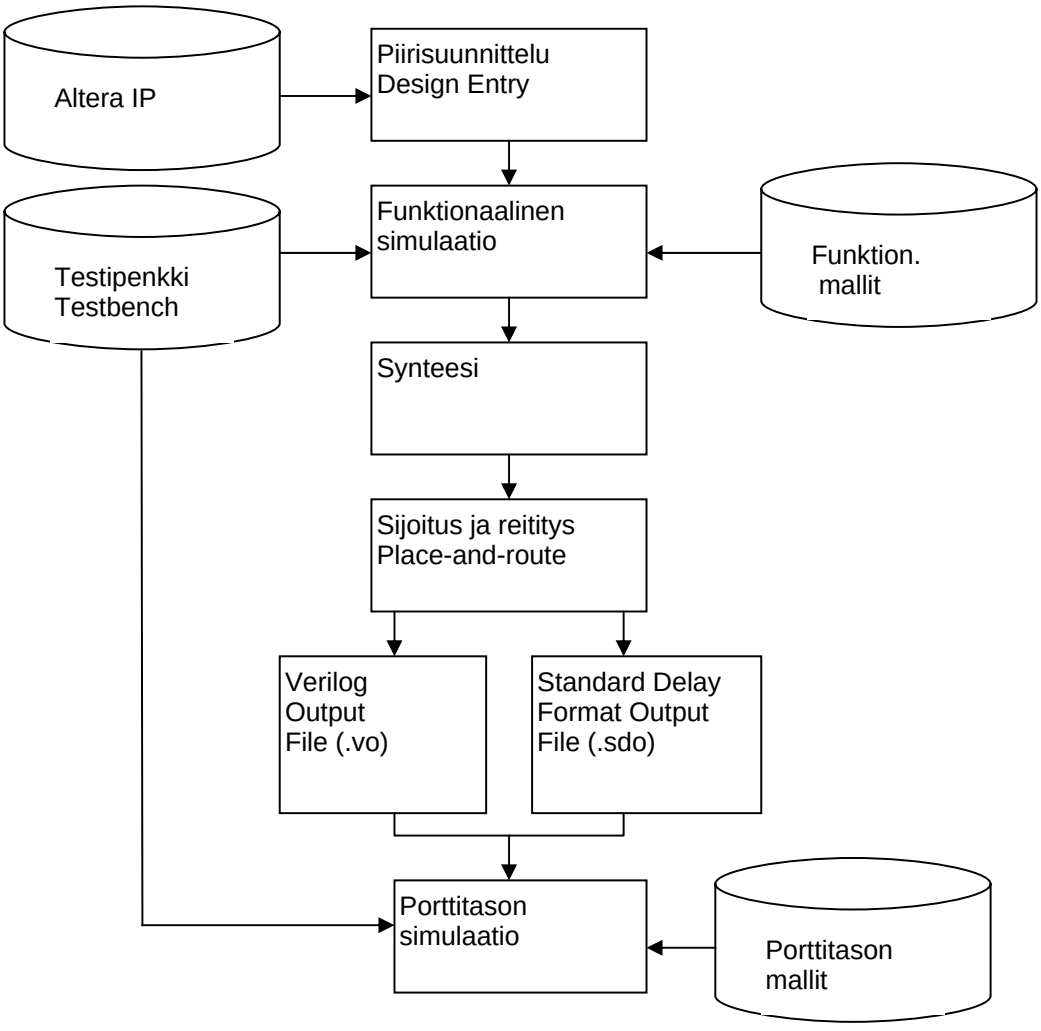
4.1.1 Simulaatio

Quartus II Simulator -simulointityökalulla voidaan suorittaa funktionaalista ja ajoitussimulaatiota. Ennen funktionaalista simulointia on generoitava funktionaalisen simulaation kytkentälista sekä Waveform-tiedosto. Simulointityökalun lisäksi Quartus II -ohjelmisto tarjoaa lisää ominaisuuksia simulaation suorittamiseen kolmansien osapuolten simulointityökalujen avulla. Esimerkkeinä työkaluista ovat Mentorin ModelSim, Synopsys VCS ja Cadence NC-Sim -

ohjelmat. Quartus II -ohjelmisto tarjoaa työkalun myös IP-lohkojen simulointiin. [30.]

Mentor Graphics ModelSim

ModelSim-Altera-ohjelmiston lisenssi sisältyy Alteran ohjelmistoon. Myös ModelSim-versiota voidaan käyttää. ModelSim-Altera-työkalu tukee VHDL- tai Verilog-muotoista toiminnallista RTL-simulaatiota sekä porttitason ajoitus-simulointia Alteran piireille. Kuvassa 14 on esitetty Altera suunnittelun vuo-kaavio käytettäessä ModelSim-ohjelmistoa. [30.]



Kuva 14. Suunnittelun vuokaavio käytettäessä Alteran Quartus II- ja ModelSim-Altera-ohjelmistoa [30]

Funktionaalinen RTL-simulaatio verifioi suunnittelun toiminnallisuuden ennen synteesin ja sijoittelu- ja reititystoiminnon suorittamista. Kun HDL-suunnittelu on verifioitu ja todettu toiminnallisuuden olevan kunnossa, voidaan siirtyä suunnittelun synteesi- ja sijoittelu- ja reititystoiminnon vaiheeseen. RTL-simulaatio tehdään ennen porttitason simulointia. [30.]

Porttitason ajoitussimulaatiossa käytetään ohjelmassa luotuja tiedostoja: sijoittelu- ja reititystoiminnon tuottama suunnittelun kytkentälista (Verilog HDL-tiedosto, vo-muotoinen tai VHDL, vho-muotoinen) sekä Standard Delay Format (SDF) -tiedosto (.sdo). Kytkentälista on tiedosto, jossa suunnitteluun on liitetty rakenteelle ominaisia perusalkioita, kuten logiikkaelementit ja I/O-elementit. SDF-tiedosto sisältää ajoitusinformaatiota jokaiselle rakenteen alkiolle ja reitityselementille (niille, jotka löytyvät vo- ja vho-tiedostoista). Porttitason ajoitussimulaatio on siis tehtävä sijoittelu- ja reititystoiminnon jälkeen, sillä se tarvitsee informaatiota siitä, miten suunnittelu on sijoitettu määrätyn laitteen rakennelohkoihin. Se verifioi suunnittelun toiminnan kun pahimman tapauksen (worst-case) ajoitusviiveet on laskettu. [30.]

Synopsys VCS

Synopsys VCS -ohjelman avulla voidaan suorittaa funktionaalista RTL-simulaatiota, synteesin jälkeistä simulaatiota sekä porttitason ajoitussimulaatiota. VCS-ohjelmassa on lisäksi graafinen debuggaus-järjestelmä, joka voidaan erikseen ottaa käyttöön sekä interaktiivinen ei-graafinen debuggausominaisuus (VCS CLI). Funktionaaliset RTL-simulaatiot verifioivat suunnittelun toiminnallisuuden ennen synteesin ja piirille sijoituksen ja reitityksen suorittamista ja ne ovat riippumattomia FPGA:n arkkitehtuuritoteutuksesta. [30.]

Synteesin jälkeinen simulaatio verifioi toiminnallisuutta synteesin suorittamisen jälkeen. Quartus II -ohjelmistossa voidaan luoda synteesin jälkeinen kytkentälista, jonka avulla suoritetaan simulaatio. Verifioinnin jälkeen kohderakenteelle suoritetaan sijoitus ja reititys. Porttitasoinen ajoitussimulaatio verifioi toiminnallisuutta sijoituksen ja reitityksen jälkeen. Quartus II -ohjelmassa luodun kytkentälistan avulla voidaan suorittaa porttitasoista ajoitussimulaatiota VCS-ohjelmassa. [30.]

Cadence NC-Sim

Cadencen NC-perheessä on useita simulaattoreita: NC-Sim, NC-Verilog ja NC-VHDL. Niiden avulla voidaan suorittaa funktionaalista ja porttitason ajoitus-simulaatioita. [30.]

Suoritettaessa simulaatiota Cadence NC -simulaattorilla käytetään suunnittelu-tiedostoja (Verilog tai VHDL). Porttitason ajoitussimulaatiolla tutkitaan portti-tason ajoitusta. Piirille sijoituksen ja reitityksen jälkeen Quartus II -ohjelmisto luo Verilog Output -tiedoston ja VHDL Output -tiedoston sekä Standard Delay Format Output -tiedoston porttitason ajoitussimulaatiota varten. [30.]

IP-lohkojen simulointi

IP (intellectual property) -lohkojen käytöllä voidaan FPGA:n suunnittelussa ly-hentää suunnittelu- ja verifiointiaikaa. IP-lohkot ovat uudelleen käytettäviä ja niitä on käytettävissä optimoituna Alteran FPGA-piirille sen suunnittelu-vaiheessa. Alteran IP Toolbench -työkalun avulla voidaan konfiguroida IP-suunnitteluja, luoda VHDL- tai Verilog-muotoinen simulaatiomalli ja käyttää si-mulointiin valittua simulointityökalua. [30.]

4.1.2 Ajoitusanalyysi

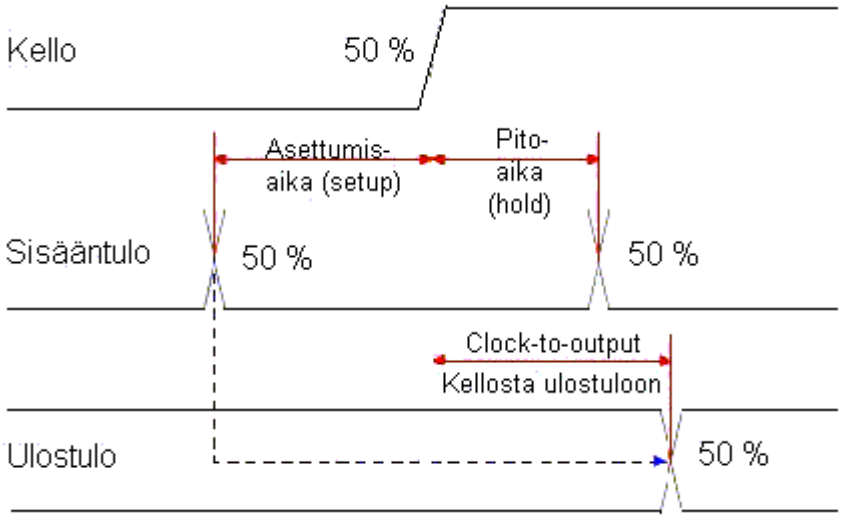
Quartus II -ohjelmisto tarjoaa mahdollisuuden suorittaa SoC-suunnittelussa ajoitusanalyysiä. Staattinen ajoitusanalyysi on keino analysoida, suorittaa de-buggausta ja osoittaa ajoituksen kelvollisuus kohteena olevassa suunnittelussa. Sen avulla mitataan jokaisen polun viive ja raportoidaan analyysin suoritusta. Ajoitusanalyysi ei kuitenkaan tarkista suunnittelun toiminnallisuutta ja sitä pitäisikin käyttää yhdessä simulaation kanssa, jotta koko suunnittelun toiminta saadaan varmennettua. [31.]

Kaikenkattavan ajoitusanalyysin tulee tutkia asettumisaikoja ja pitoaikoja, vii-veitä kellotulosta lähtöön, enimmäiskellotaajuuksia ja suunnitteluun sisältyviä ”löysiä” aikoja. Saadun tiedon perusteella voidaan varmentaa piirin toiminta ja

löytää mahdolliset aikarikkomukset. Ajoitukseen liittyvät rikkomukset voivat löytymättöminä aiheuttaa virhetilanteita piirin toiminnassa. [31.]

Kellon asettumisaika (setup time t_{su}) ja pitoaika (hold time t_h)

Jotta kiikku toimisi luotettavasti, on valmistelutulojen oltava muuttumatta kellotulon aktiivisen reunan kohdalla. Valmistelutulojen on oltava vakioita asettumisajan verran ennen kellotulon muutosta ja pitoajan verran sen jälkeen (kuva 15) [11].



Kuva 15. Asetus- ja pitoaika sekä kello-ulostulo-viive [31]

Asettumisaika on minimiaika, joka datan on pysyttävä muuttumattomana ennen aktiivista kellon reunaa. Pitoaika on vastaavasti minimiaika, joka datan on pysyttävä vakiona aktiivisen kellon reunan jälkeen. Rekistereillä on lisäksi myös omat sisäiset pito- ja asettumisaikansa. [31.]

Kellosta ulostuloon -viive (clock-to-output delay, t_{co}) (kuva 15) on enimmäisaika, joka sallitaan, jotta saavutetaan kelvollinen ulostulo sen jälkeen kun kello on vaihtanut tilaa rekisterin kellosisääntulossa. Rekisterillä on oma sisäinen t_{co}-aika. [31.]

Nastasta nastaan (pin-to-pin) -viive t_{pd} on aika, joka vaaditaan sisääntulonastan signaalilta, jotta se ehtii kulkea kombinaatiologiikan läpi ja saapua

ulostulonastaan. Quartus II -ohjelmistossa voidaan tehdä t_{pd}-osoituksia sisääntulonastan ja rekisterin väliin, kahden rekisterin väliin ja rekisterin ja ulostulonastan väliin. [31.]

Maksimikellotaajuus (f_{max}) on suurin nopeus, millä suunnittelukello toimii haittaamatta sisäisiä asetus- ja pitoaikavaatimuksia. Nopeus riippuu pisimmästä viiveestä, joka esiintyy millä tahansa polulla kahden saman kellon ajoittaman rekisterin välillä. Quartus II -ohjelma voi suorittaa ajoitusanalyysiä sekä yhden että useampia kelloja sisältävissä suunnitelmissa. [31.]

Vaaditun ja todellisen kellon aikajakson erotusta kutsutaan nimellä ”slack” (välys). Se on erotus, millä ajoitusvaatimukset saavutetaan tai ei saavuteta. Positiivinen arvo ilmoittaa eron, jolla vaatimukset täytettiin, negatiivinen arvo taas eron, jolla niitä ei täytetty. Käytössä on myös pitoaikavälys, joka ilmoittaa eron, millä pitoaikavaatimusten minimiarvot saavutetaan tai ei saavuteta. Ohjelma ilmoittaa kuitenkin vain poluissa tapahtuneista pitoaikarikkomuksista negatiivisilla arvoilla. [31.]

Quartus II -ohjelmisto suorittaa täyden käännöksen yhteydessä automaattisesti ajoitusanalyysin, jonka tulokset esitetään käännösraportissa. Timing Analyzer -työkalun avulla voi tehdä erityisiä asetuksia globaalisti koko projektille tai yksittäisille alueille. Yksittäiset ajoitusasetukset ovat etusijalla jos kummallekin on annettu oma määritys. Globaaleja vaatimuksia clock-to-output-viiveille, nastasta nastaan -viiveille ja asettumisaikaviiveille sekä minimivaatimuksia pitoajan, clock-to-output-viiveen ja nastasta nastaan -viiveen suhteen voidaan asettaa. Globaali enimmäiskellotaajuuden vaatimus tai yksittäisten kellojen suhteet ja ajoitusvaatimukset voidaan myös määrittää. Kellot voidaan määrittää toisistaan riippuviksi tai riippumattomiksi. [31.]

Yksittäisille pinneille ja solmuille voidaan tehdä ajoituksellisia määrittelyasetuksia. Sisääntulon maksimiviive -asetuksen avulla voidaan määrittää suurin sallittu viive signaalille laitteen ulkopuolisesta ulkoisesta rekisteristä johonkin määrättyyn sisääntuloon. Pienin sallittu viive on samoin asetettavissa sisääntulon minimiviive -asetuksella. Ulostuloviiveiden määrityksissä on vastaavat asetusmahdollisuudet. [31.]

Yksittäisiä t_{co}-, t_h-, t_{pd}- *tai* t_{su}-asetuksia tehtäessä ne menevät globaalien asetusten edelle. Asetus t_{co}:lle voidaan tehdä nastalle, ulostulorekisterille tai ulostulorekisteristä nastaan. Yksittäinen t_h-asetus voidaan tehdä joko nastalle, sisääntulorekisterille, nastasta sisääntulorekisteriin tai kellonastasta sisääntulorekisteriin. Asetus t_{pd}:lle voidaan tehdä sisääntulonastoista ulostulonastoihin, sisääntulonastoista rekistereille, rekistereistä rekistereille, rekistereistä ulostulonastoihin tai pelkälle sisääntulonastalle. Tehtäessä t_{su}-asetuksia ne voidaan kohdentaa sisääntulonastaan, sisääntulorekisteriin, sisääntulonastasta sisääntulorekisteriin tai kellonastasta sisääntulorekisteriin. [31.]

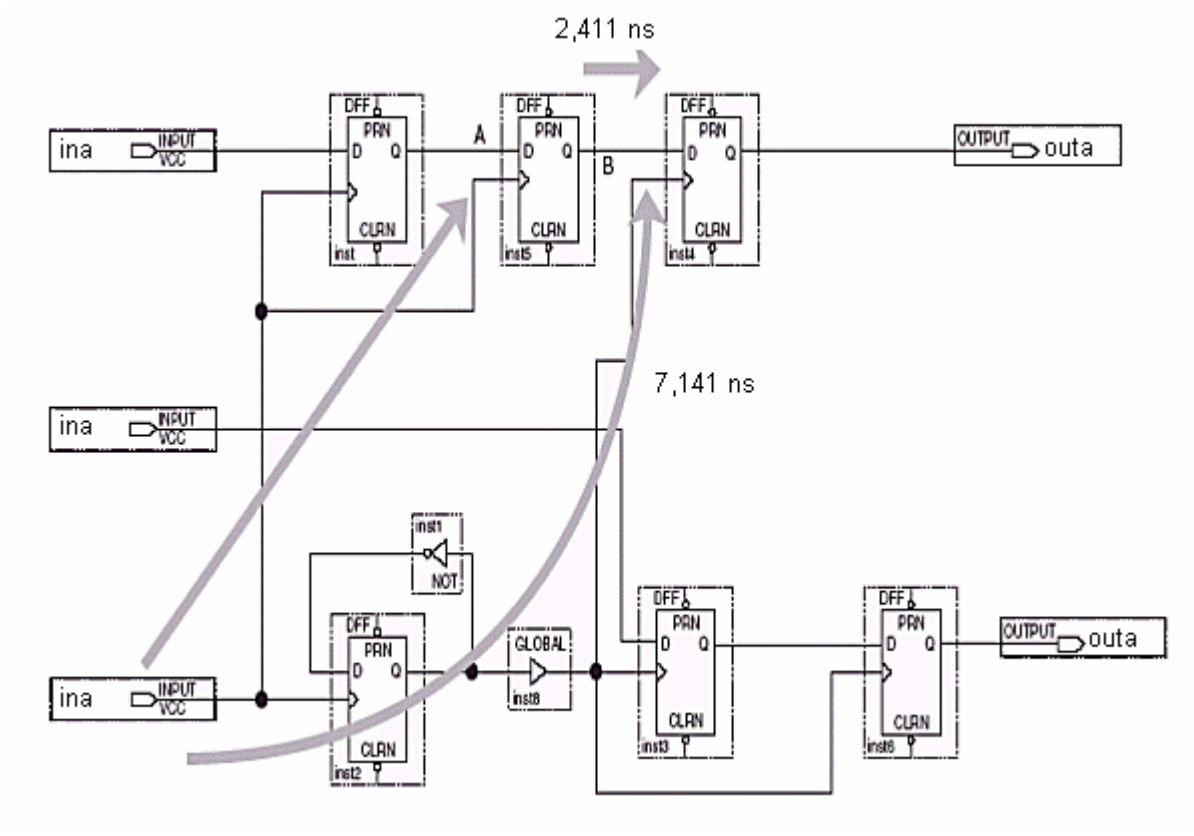
Quartus II -ajoitusanalyysin raportissa on esitettynä f_{max} ja ”slack” kaikille kellopinneille. Se näyttää myös clock-to-output-viiveet kaikille ulostulopinneille, asettumis- ja pitoajat sisääntulopinneille ja nastasta nastaan -viiveet kaikille nastasta nastaan -yhdistelmien poluille. Negatiivinen slack-arvo ilmoittaa polun epäonnistumisen kelloajoitusten suhteen ja positiivinen arvo taas sen, millä arvolla polku ”voittaa” vaatimukset. Jos jokin pitoaika, asettumisaika sekä clock-to-output on määritelty halutulla vaatimuksella eikä globaaleja määrityksiä niille ole, on raportissa esitetty vain yksittäiset tulokset. Em. arvot kaikille I/O-pinneille on kuitenkin mahdollista saada globaaleilla määrityksillä. [31.]

Kehittyneempi ajoitusanalyysi

Kehittyneempää ajoitusanalyysiä voidaan suorittaa Quartus II -ohjelmiston avulla suunnitteluissa, joissa on kellon hallinta-alueen poikki meneviä polkuja tai monijakso-ohjauspolkuja. Monijakso-ohjauspolut ovat sellaisia datapolkuja rekistereiden välissä, että ne tarvitsevat enemmän kuin yhden kellojakson lukitakseen datan kohderekisteriin. Eräs tilanne vaativampien ominaisuuksien käytöstä on esitetty seuraavassa esimerkissä. [31.]

Eroavuudet kellosignaalien saapumisajoissa

Ajoitusanalyysin raportissa voi olla mainintoja eroavuuksista kellosignaalin saapumisajoissa eri rekistereille (clock skew). Kuvassa 16 on esitetty esimerkki tällaisesta tilanteesta. Suunnittelussa on kellohaaroja ja erittäin lyhyt datapolku kahden rekisterin välillä. [31.]



Kuva 16. Erot kellosignaalien saapumisajoissa. Kahden rekisterin välinen datapolku on erittäin lyhyt. [31.]

Kuvassa 16 on nähtävissä pisin polku (7,141 ns) kellosta a rekisteriin inst4. Lyhin datapolku (1,847 ns) on kellosta a rekisteriin inst5. Kellon vääristymäksi saadaan polkujen ajan erotus, joka on 5,294 ns. Lyhin rekistereiden välinen datapolku lähderekisterin ja määränpään rekisterin välillä on 2,411 ns. Täten kellon vääristymän arvo (5,294 ns) on suurempi kuin datapolku (2,411 ns). Seurauksena on virheellinen tilanne piirin toiminnallisuudessa. Jos tilanne halutaan korjata, on polku B (rekistereiden välinen) pidennettävä lisäämällä polkuun soluja tai sijoittelemalla uudelleen lähde- ja kohderekisterit. [31.]

Parhaan tapauksen ajoitusanalyysi

Parhaan tapauksen ajoitusanalyysi (best case timing analysis) tunnetaan myös nimellä ”minimi ajoitusanalyysi”. Se mittaa ja raportoi minimin kellosta ulostuloon -viiveen (t_{co}), minimin nastasta nastaan -viiveen (t_{pd}), pitoajan (t_h) ja kellon pidon (clock hold). Parhaan tapauksen ajoitusanalyysi suoritetaan tarkistamalla minimit viivevaatimukset parhaan tapauksen viiveen ajoitusmalleista. Mallit luonnehtivat laitteen toiminnan suurimmalla jännitteellä,

nopeimmalla prosessilla ja matalimmilla lämpöolosuhteilla. Huonoimman tapauksen mallit taas luonnehtivat laitteen toiminnan, kun prosessi on hitaimmillaan, lämpötilat korkeimmillaan ja jännite alimmillaan. Vakiomuotoisessa ajoitusanalyysissä raportoidaan myös minimiviiveen tarkastus, kuten pitoaika, käyttäen huonoimman tapauksen mallia. Käyttäjä voi tehdä globaaleja määrityksiä minimiviiveille, kuten myös yksittäisiä minimiajoitusasetuksia suunnittelun pinneille ja rekistereille. [31.]

Aikainen ajoituksen arviointi

Suuri osa Quartus II -ohjelmiston käännöstapahtumasta kuluu sijoituksen ja reitityksen algoritmeihin, joita käytetään optimaalisen suunnittelun tuloksen saamiseen. Laajemmilla suunnitteluilla voidaan suunnitteluprosessia nopeuttaa aikaisen ajoituksen arvioinnin (Early Timing Estimate) ominaisuuden avulla. Toimenpide antaa arvion ajoituksesta. Arvio on kuitenkin melko lähellä lopullista, normaalissa käännöstapahtumassa saatua arvoa. [31.]

4.1.3 Tehon arviointi ja analysointi

Tehonkulutus on käymässä yhä tärkeämmäksi aiheeksi suurempien suunnittelujen ja prosessiteknologian myötä. Kortin suunnittelussa on oltava jo selvillä tarkka arvio laitteen kuluttamasta tehosta, jotta voidaan suunnitella tehonkulutus, virtalähteet, jänniteregulaattorit, jäähdytyslevyt ja jäähdytysjärjestelmä. Stratix II -piirin kulutus voidaan arvioida käyttämällä Microsoftin Excel-pohjaista taulukkolaskentaohjelmaa PowerPlay Early Power Estimator. PowerPlay Early Power Estimator -taulukkolaskentaohjelmaa käytetään kortin suunnittelussa tehon arviointiin ja järjestämiseen mahdollisimman hyvin. Arvio perustuu tyypillisiin olosuhteisiin. [32.]

Tehoon liittyviä ominaisuuksia voidaan arvioida eri vaiheissa suunnittelua. Suunnittelun vaiheen ja vaaditun tarkkuuden perusteella voidaan valita käytetäänkö tehon arviointiohjelmaa vai analysointia. Useinhan FPGA:n suunnittelun yhteydessä pyritään nopeuteen, ja kortin ja laitteen tehon suunnittelun on oltava valmiina ennen kuin FPGA-suunnittelu valmistuu. Ennen suunnitelman valmis-

tusta voidaankin käyttää tehon arvioinnin ohjelmaa Early Power Estimator. Quartus II -ohjelmiston luomaa tehonarviointitiedostoa voidaan käyttää apuna datan lisäämisessä ohjelmaan. [32.]

Quartus II -ohjelmistossa on myös suunnittelun analysointiin PowerPlay Power Analyzer -työkalu, jota käytetään suunnitelman ollessa valmis. Ohjelman avulla voidaan varmistaa, että lämpötiloihin ja teholähteisiin liittyvät vaatimukset ovat kunnossa ja sen avulla voidaan hioa tehon arviointia. Ohjelmaa käytetään synteesin sekä sijoitus- ja reititystoiminnon jälkeen. [32.]

4.1.4 Virheiden haku ja korjaus (debuggaus)

Verifiointityökaluja tarvitaan läpi koko FPGA:n suunnittelun. Kun ohjelma on otettu käyttöön, implementoitu, voidaan suunnittelusta etsiä virheitä normaaleissa käyttöolosuhteissa ja lopullisessa tuotteessa käytetyillä nopeuksilla. Quartus II -ohjelmistossa on implementoinnin jälkeen käytettäviä työkaluja, joita ovat SignalProbe ja SignalTap II Logic Analyzer. Niiden avulla voidaan analysoida laitteen sisäisiä solmukohtia ja I/O-pinnejä. SignalTap II Logic Analyzer käyttää sulautettua logiikka-analysaattoria signaalin reitittämiseen JTAG-liitännän kautta joko SignalTap II logiikka-analysaattoriin tai ulkoiseen logiikka-analysaattoriin tai oskilloskooppiin. SignalProbe -ominaisuuden avulla saadaan myös signaali reititettyä ulkoisiin laitteisiin, oskilloskooppiin tai logiikka-analysaattoriin. Kolmantena Quartuksen ominaisuutena on Chip Editor, jota käytetään yhdessä SignalTap II- ja SignalProbe -työkalujen virheenhaun apuna. [33.]

Signal Probe

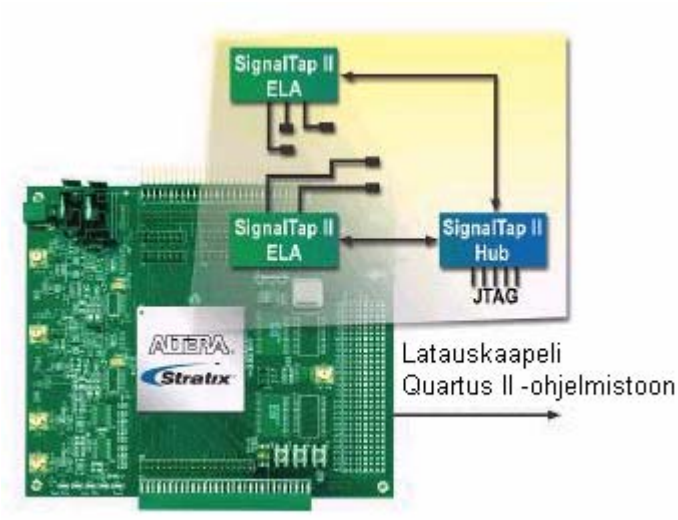
Signal Probe -työkalun avulla voidaan reitittää käyttäjän määrittelemät sisäiset signaalit I/O-pinneihin vaikuttamatta suunnitteluun. Sen avulla saadaan tehtyä samat toiminnot kuin Chip Editor -työkalullakin. Signal Probe -työkalun avulla tehty reititys ei vaikuta lähteen käyttäytymiseen tai suunnittelun toimintaan. Valittuja signaaleja voidaan debugata ilman täyttä käännöstapahtumaa. Signal

Probe -käännös voidaan suorittaa. Se yhdistää signaalit käyttämättömiin tai ennalta varattuihin pinneihin ja sitä kautta ulkoiseen logiikka-analysaattoriin. [34.]

SignalTap II Logic Analyzer

SignalTap II Logic Analyzer arvioi suunnittelun sisäisten signaalien tiloja päämääränä suunnitteluvirheiden hakeminen. Se käyttää sulautettua logiikka-analysaattoria reitittämään signaalin dataa JTAG-liitännän kautta joko SignalTap II Logic Analyzer -työkaluun, ulkoiseen logiikka-analysaattoriin tai oskilloskooppiin. Ohjelmallisesti toteutettu sulautettu logiikka-analysaattori on käyttökelpoinen vaikkapa silloin kun piirien pienen koon ja nastojen tiheyden takia ei voida käyttää perinteistä logiikka-analysaattoria. [35.]

Jotta voitaisiin suorittaa logiikka-analyysiä SignalTap II sulautetulla logiikka-analysaattorilla, tarvitaan testattavan laitteen lisäksi Quartus II -ohjelmisto ja latauskaapeli. Data kerätään näytekellon nousevalla reunalla ja tallennetaan laitteen muistilohkoihin. JTAG-liitännän ja latauskaapelin (EthernetBlaster tai USB Blaster™) avulla data siirretään Quartus-ohjelmistoon näytettäväksi aaltomuotoisena tietokoneen näytöllä. Kuvassa 17 on esitetty SignalTap II -logiikka-analysaattorin lohkokaavio. [35.]

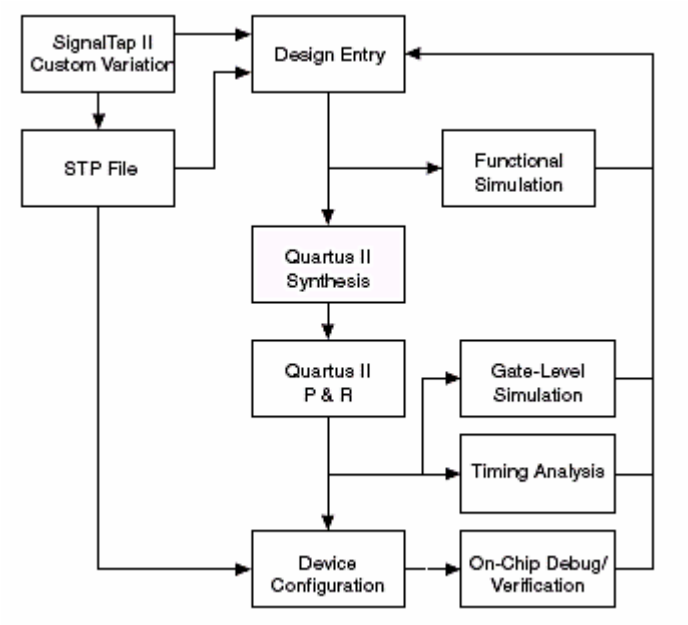


Kuva 17. SignalTap II sulautetun logiikka-analysaattorin lohkokaavio [35]

SignalTap II sulautetun logiikka-analysaattorin ominaisuuksia ovat [35]:

- Yhteen piiriin voidaan liittää useita logiikka-analysaattoreita. (Dataa voidaan kerätä useiden kellojen hallinta-alueilta.)
- Useita logiikka-analysaattoreita voidaan liittää useisiin piireihin yhdessä JTAG-ketjussa. (Yhtäaikaisesti voidaan kerätä dataa useilta JTAG-ketjun piireiltä.)
- Jokaiselle analysaattorille voidaan asettaa useita perus- tai kehittyneempiä liipaisutasoja. (Monimutkaisempia käskyjä voidaan antaa analysaattorille.)
- Jokaisessa laitteessa 1024 kanavaa. (Signaaleja ja leveämpiä väylärakenteiden signaaleja voidaan kerätä.)
- Kanavalta voidaan kerätä max. 128 k näytettä.
- Dataa voidaan kerätä max. 200 MHz nopeudella.
- Analysaattorin käyttämien konfiguraatioiden aiheuttama resurssien (logiikka ja muisti) käytön arviointi. [35.]

SignalTap II Logic Analyzer -työkalua voidaan käyttää luomalla erityinen SignalTap II -tiedosto (.stp) ja konfiguroimalla sen yksityiskohdat (kello datanäytteitä varten, signaalit, talletettujen näytteiden määrä joka signaalille, analysaattorin liipaisu, liipaisua edeltävän datan määrä). Stp-tiedosto voidaan luoda ja konfiguroida myös MegaWizard Plug-In Manager -työkalun avulla ja siirtää luotu HDL-moduuli suunnittelun HDL-koodiin. Stp-tiedoston avulla saadaan luotua sulautettu logiikka-analysaattori käytettäväksi suunnittelussa. Tiedosto sisältää asetukset analysaattorille sekä kerätyn datan analysointia ja esittämistä varten. Kuvassa 18 on esitetty prosessi, jossa esitetään SignalTap II Logic Analyzer -työkalun käyttöönotto ja käyttö molemmilla tavoilla. [35.]



Kuva 18. SignalTap II sulautetun logiikka-analysaattorin vuokaavio [35]

Kun stp-tiedosto on konfiguroitu, se on käännettävä Quartus II -projektin kanssa, jotta sitä voitaisiin käyttää suunnittelun analysoimiseen. Kun stp-tiedosto ensimmäisen kerran luodaan, se lisätään automaattisesti projektiin. Sen voi kuitenkin lisätä myös manuaalisesti. Käännöstoiminnon jälkeen voidaan FPGA ohjelmoida käytettäväksi SignalTap II -logiikka-analysaattorin kanssa. [35.]

Kun analysaattoria käytetään, se ajaa ohjelmaa eri vaihtoehdoilla [35]:

- Run: Ohjelmaa ajetaan kunnes liipaisu tapahtuu, datan keräys pysähtyy.
- Stop: Analyysi pysäytetään. Saatu data ei ilmesty näkyviin jos liipaisua ei ole tapahtunut.
- Auto Run: Dataa kerätään keskeytymättä, kunnes Stop-nappia painetaan.
- Read Data: Kerätty data esitetään. [35.]

SignalTap II Logic Analyzer -työkalussa on useita lisäominaisuuksia debuggauksen suorittamiseen. Työkalun avulla voidaan konfiguroida analysaattori tallentamaan kerätty data laitteen RAM-muistiin. Jos muistin koko on rajoitettu voidaan data reitittää käyttämättömiin I/O-pinneihin analysoitavaksi ulkoisella logiikka-analysaattorilla. Tällöin ohjelma luo automaattisesti debuggausporttisignaalit, jotka yhdistävät sisäiset FPGA-signaalit ulkoisiin pinneihin. Voidaan myös luoda liipaisun sisääntulo, jossa liipaisu suoritetaan ulkoisesta läh-

teestä, tai liipaisun ulostulo, jossa saadaan ulkoinen signaali liipaisemaan muita laitteita. Tämä on tarpeellista, jos halutaan synkronoida työkalu ulkoisen logiikka-analysaattorin kanssa. Muutoinkin työkalu tarjoaa mahdollisuudet monipuolisten liipaisuolosuhteiden luomiseen. [35.]

Jokaiselle suunnittelun kellon hallinta-alueelle voidaan luoda oma logiikka-analysaattori SignalTap II Logic Analyzer -työkalun avulla. Lisäksi voidaan tehdä erilaisia asetuksia signaaliryhmille saman kellon alueella. Jos osalle signaaleista käytetään suurempaa näytemäärää, esimerkiksi 64 000, voidaan toiselle signaalijoukolle määritellä vaikkapa 1000 näytteen näytemäärä. [35.]

SignalTap II -logiikka-analysaattoriin halutaan mahdollisesti tehdä jälkikäteen muutoksia, kuten esimerkiksi muuttaa näytteistysmäärää tai lisätä uusia signaaleja tutkittaviksi. Työkalussa on ominaisuus, Incremental Routing, jonka avulla muutokset voidaan tehdä tarvitsematta suorittaa täyttä käännöstapahtumaa. [35.]

Kerätty data voidaan kääntää muihin tiedostomuotoihin. Tiedostoja voidaan sitten käyttää muiden valmistajien simulointityökalujen kanssa. Esimerkkeinä tiedostomuodoista ovat graafiset muodot (.jpg ja .bmp) sekä .vwf (Vector Waveform file) -muoto. [35.]

Käytetty RAM-tyyppi voidaan valita kerätyn datan tallentamiseen. Jos suunnittelussa toteutetaan suuri puskurointisovellus, kuten on esimerkiksi järjestelmän välimuisti, voidaan sovellus sijoittaa M-RAM-lohkoon ja jäljelle jäävät M512- tai M4K-lohkot voidaan jättää analysaattori-työkalun käyttöön. [35.]

Analysaattorissa on ominaisuus, jolla SignalTap II -analysaattorin käyttämät FPGA:n resurssit arvioidaan. Resurssiarviointi laskee jokaisen analysaattorin käyttämän muistin määrän sekä käytetyt logiikkaelementit. Laitteen resurssit saattavat olla rajoitetut ja on ehkä tarpeen tietää, mitkä niistä ovat analysaattoreiden käytössä. [35.]

Suunnittelusta virheitä haettaessa saattaa käytössä olla useampi stp-tiedosto. Jokaisessa tiedostossa on erilaisia signaaliryhmiä, joilla virheentarkistusta suo-

ritetaan suunnittelun eri lohkoille. Jokaiseen stp-tiedostoon on yhdistetty ohjelmointitiedosto (.sof). SignalTap II -työkalussa on ominaisuuksia useiden stp-tiedostojen hallintaan (Data Log, SOF Manager). [35.]

SignalTap II Logic Analyzer -työkalua voidaan käyttää myös muualla sijaitsevan laitteen debuggaukseen. Etäkäyttöä varten tarvitaan Quartus II -ohjelmisto asennettuna paikalliseen tietokoneeseen ja SignalTap II Logic Analyzer -työkalun itsenäinen versio (Stand-alone SignalTap II) asennettuna etäkoneeseen. Etäkohteessa on piirilevylle oltava kytkettynä ohjelmointilaitteisto (esim. USB-Blaster™ tai ByteBlaster™). Lisäksi tarvitaan TCP/IP-yhteys kohteiden välille. Etäällä oleva piiri ohjelmoidaan paikalliselta koneelta TCP/IP-yhteyttä ja etäkoneella olevaa ohjelmointilaitteistoa käyttäen. Virheiden korjaus on myös mahdollista, sillä piiri voidaan päivittää verkon kautta. [35.]

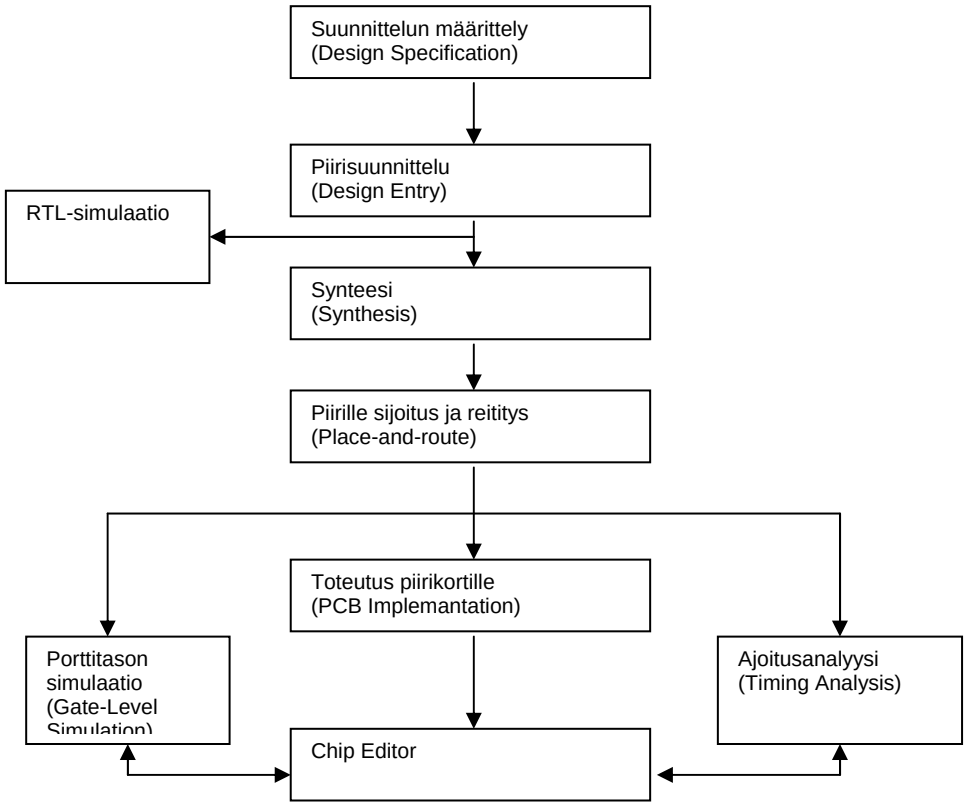
Kun käännetään projekti, johon sisältyy SignalTap II -logiikka-analysaattori, lisätään suunnitteluun samalla IP-lohko. Mahdollista olisi, että lisäyksellä olisi vaikutus suunnittelun sijoituksiin ja reitityksiin sekä ajoitukseen. Vaikutusten minimoimiseksi voidaan tehdä esim. back-annotate-toiminto ennen analysaattorin lisäystä ohjelmaan. [35.]

Jos suunnittelusta on löytynyt virhe, voidaan signaali paikallistaa. Virhe voidaan korjata käyttämällä Chip Editor -työkalua. [35.]

Chip Editor

Ideaalitapauksessa suunnitteluvuo alkaa suunnittelun spesifikaatioiden kehittämisestä. Seuraavaksi luodaan rekisterinsiirtotasoinen (RTL) koodi ja verifioidaan, että RTL-koodi suorittaa halutun toiminnallisuuden. Verifioidaan myös, että sijoitettu ja reititetty suunnittelu tyydyttää ajoitukselle asetetut vaatimukset. Lopuksi ohjelmoidaan käytettävä FPGA. Usein suunnitteluvuossa esiintyy ongelmia. RTL-koodissa on virheitä tai määrittelyjä vaihdetaan. Perinteisesti on mentävä takaisin RTL-koodiin ja tehtävä vaaditut muutokset. Seuraavaksi on käytävä läpi koko suunnitteluvuo uudelleen, tehtävä synteesi, piirille sijoitus ja reititys sekä verifiointi. [36.]

Quartus II -ohjelmistossa on työkalu Chip Editor, jonka avulla voi nähdä Alteran laitteen sisäisen rakenteen ja editoida toiminnallisuutta tai parametrien asetuksia. Muutoksia tehtäessä ei tarvita täyttä käännöstapahtumaa. Chip Editor -työkalu toimii piiriin sijoituksen ja reitityksen jälkeisten tietokantojen kanssa. Muutokset tehdään suoraan sijoitus ja reititys -kytkentälistaan, generoidaan uusi ohjelmointitiedosto ja testataan muutettu suunnittelu suorittamalla porttitasoinen ajoitussimulaatio ja ajoitusanalyysi RTL-koodia muuttamatta. Muutosten tekoa voidaan jatkaa kunnes ongelma on korjattu. Koska tehdyt muutokset rajoittuvat tiettyihin laitteen resursseihin, ei muiden osien ajoituksiin tule muutoksia. Suunnittelun säännöt tarkastetaan muutosten yhteydessä suunnitteluvirheiden varalta. Kuvassa 19 on esitetty suunnitteluvuo, jossa on mukana Chip Editor -työkalu. [36.]



Kuva 19. Chip Editor suunnitteluvuossa [36]

Chip Editor -työkalun avulla voidaan visuaalisesti nähdä esimerkiksi miten suunnittelun lohkot on yhdistetty sekä mitkä signaalit yhdistävät lohkoja. Logiikkaelementtien (LE) konfiguroinnit ovat nähtävillä. Mukautuvat

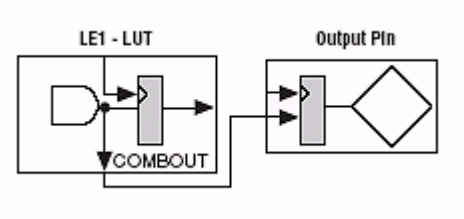
logiikkamoduulit (ALM) ja niiden konfigurointi suunnittelussa on myös nähtävissä. I/O-resurssien käyttö, signaalitiet ja muut vastaavat seikat ovat nähtävissä, samoin PLL-komponettien konfigurointi. Chip Editor -työkalulla voidaan tehdä muutoksia logiikkaelementeille, mukautuville logiikkamoduuleille, I/O-soluille, PLL-komponenteille, elementtien välisille yhteyksille ja elementtien sijoituksille. [36.]

Stratix II -arkkitehtuurin logiikan peruslohko on mukautuva logiikkamoduuli, ALM. Jokainen mukautuva logiikkamoduuli sisältää valikoiman hakutaulukkopohjaisia resursseja, jotka voidaan jakaa kahden mukautuvan hakutaulukon (ALUT) kesken. Joka mukautuva logiikkamoduuli voi toteuttaa useita eri kahden funktion vaihtoehtoja kahden mukautuvan hakutaulukon sisääntulojen avulla. Hakutaulukkopohjaisten resurssien lisäksi jokaisella mukautuvalla logiikka-moduulilla on kaksi ohjelmoitavaa rekisteriä, kaksi summainta, siirtoketju, jaettu aritmetiikkaketju ja rekisteriketju. Resurssien avulla mukautuva logiikkamoduuli voi toteuttaa aritmeettisia funktioita sekä siirtorekistereitä. [13.]

Chip Editor -työkalussa on käytettävissä pohjapiirros (floorplan), jossa ovat nähtävissä käännöksen jälkeiset sijoitukset, yhteydet ja reitityspolut. Näkymä toimii eri hierarkkisilla tasoilla. Zoomaustason kasvaessa abstraktiotaso laskee (ja samalla enemmän yksityiskohtia on nähtävissä). Esim. kolmannella näky-män tasolla voidaan liikutella logiikkaelementtejä ja I/O-osia eri fyysisiin paik-koihin sekä niitä voidaan luoda kokonaan uusia. [36.]

Chip Editor -työkalun ominaisuuksiin kuuluu resurssiominaisuuseditori (Resource Property Editor). Editorin avulla voidaan katsella ja editoida mm. mu-kautuvia logiikkamoduulilohkoja. [36.]

Tyypillisesti Chip Editor -työkalun avulla suoritetaan toiminnallisten virheiden korjausta suunnittelussa, reititetään sisäinen signaali ulostulonastaan tai sää-detään PLL:n vaihesiirto vastaamaan I/O:n ajoitusta. Signaalin reitityksellä käyttämättömään ulostuloon saadaan kaapattua sisäinen signaali ulkoisen logiikka-analysaattorin avulla (kuva 20). [36.]



Kuva 20. Sisäisen signaalin reitittäminen ulostulonastaan. Luodaan ulostulonasta. Lähteenä olevaan logiikkaelementtiin (LE) luodaan ”combout” ja yhdistetään se nastan sisääntuloon. [36.]

Muistin päivitys

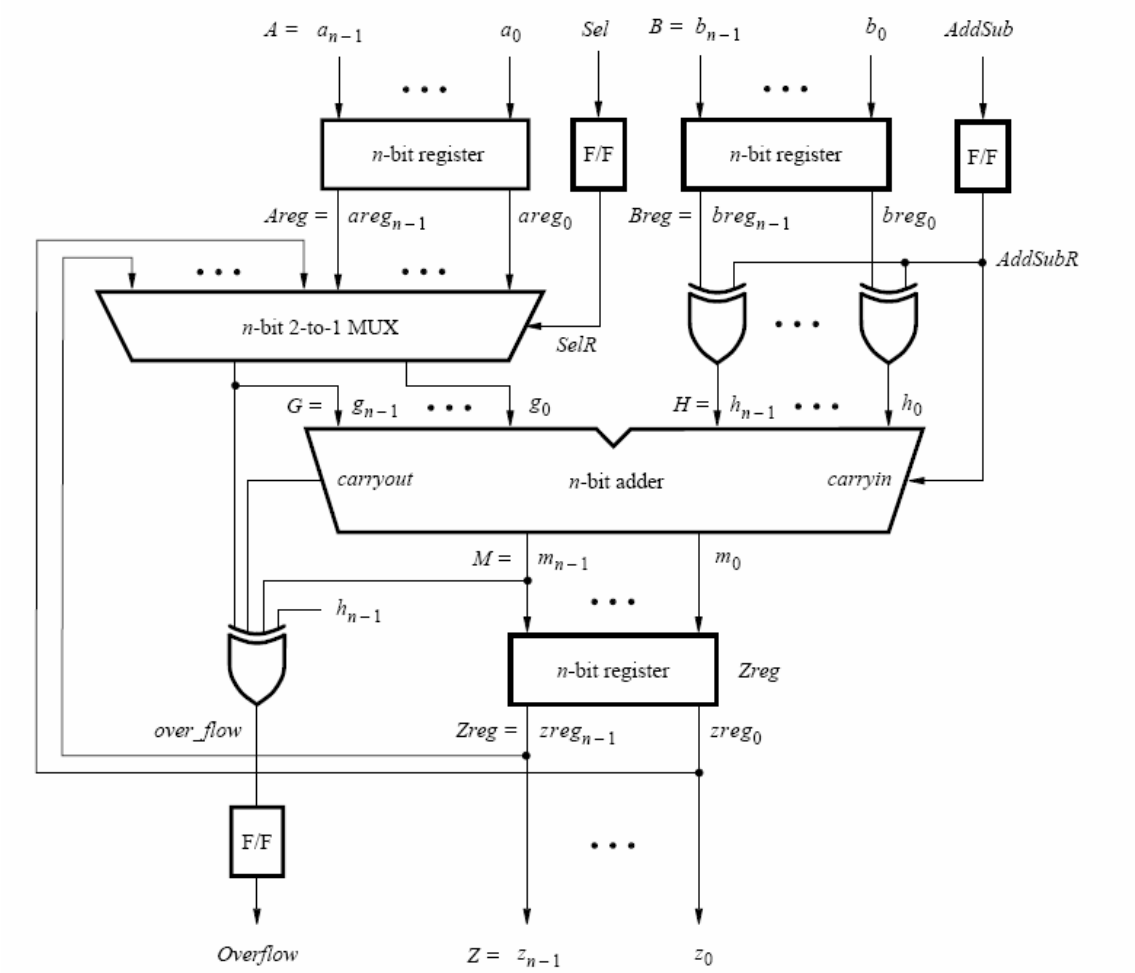
Quartus II -ohjelmistossa on mukana In-System Memory Content Editor -työkalu muistin ja vakioiden päivitykseen kun FPGA on jo toiminnassa piirilevyllä. Päivitysominaisuus tarjoaa pääsyn kirjoittaa ja lukea FPGA:n muisteja ja vakiota JTAG-liitännän kautta. Dataa voidaan lukea muisteista ja vakioista, jolloin ongelmien lähteitä voidaan tutkia. Kirjoittamisominaisuuden avulla voidaan ohittaa toiminnallisia kysymyksiä kirjoittamalla odotetun kaltainen data. Työkalun avulla voidaan vaikkapa korjata muistin väärä pariteettibitti kirjoittamalla oikea arvo RAM:iin samalla kun järjestelmä jatkaa toimintaansa. Päivitysominaisuutta voidaan käyttää yhdessä SignalTap II sulautetun logiikka-analysaattorin kanssa. Niitä on kuitenkin käytettävä yhtäaikaisesti, sillä molemmat työkalut käyttävät JTAG-liitäntää. [37.]

4.2 Formaali verifiointi

Quartus II -ohjelmiston kanssa ovat käytettävissä formaaliin, systemaattiseen verifiointiin muiden valmistajien työkalut kuten Cadence Incisive Conformal ja Synplicity Synplify -työkalut. Lähteenä olevien suunnittelutiedostojen ja Quartus II -ohjelmiston ulostulevien tiedostojen loogista vastaavuutta todennetaan. Formaali verifiointi käyttää matemaattisia tekniikoita suunnittelun toiminnallisuuden verifiointiin. Vastaavuustarkistus ja mallitarkastus ovat kaksi formaalisen verifioinnin tyyppiä. Tällä hetkellä ei Stratix II -piirille ole tukea formaalin verifioinnin suorittamiseen Quartus II -ohjelmistossa. [38.]

4.3 Esimerkki suunnittelun verifioinnista Quartus II -ohjelmiston avulla

Quartus II -ohjelmiston verifiointityökalujen käytöstä toteutettiin esimerkki. Esimerkin avulla tutustuttiin verifiointityökalujen toimintaan suunnittelun eri vaiheissa. Käytössä oli ohjelmiston versio 4.2. Esimerkiksi valittiin Quartus II -opetusohjelmiston Verilog-koodilla toteutettu vähennys- ja yhteenlaskua toteuttava suunnitelma (kuva 21) [39]. Suunnitelman (”addersubtractor”) Verilog-kielinen koodi on esitetty liitteessä A.



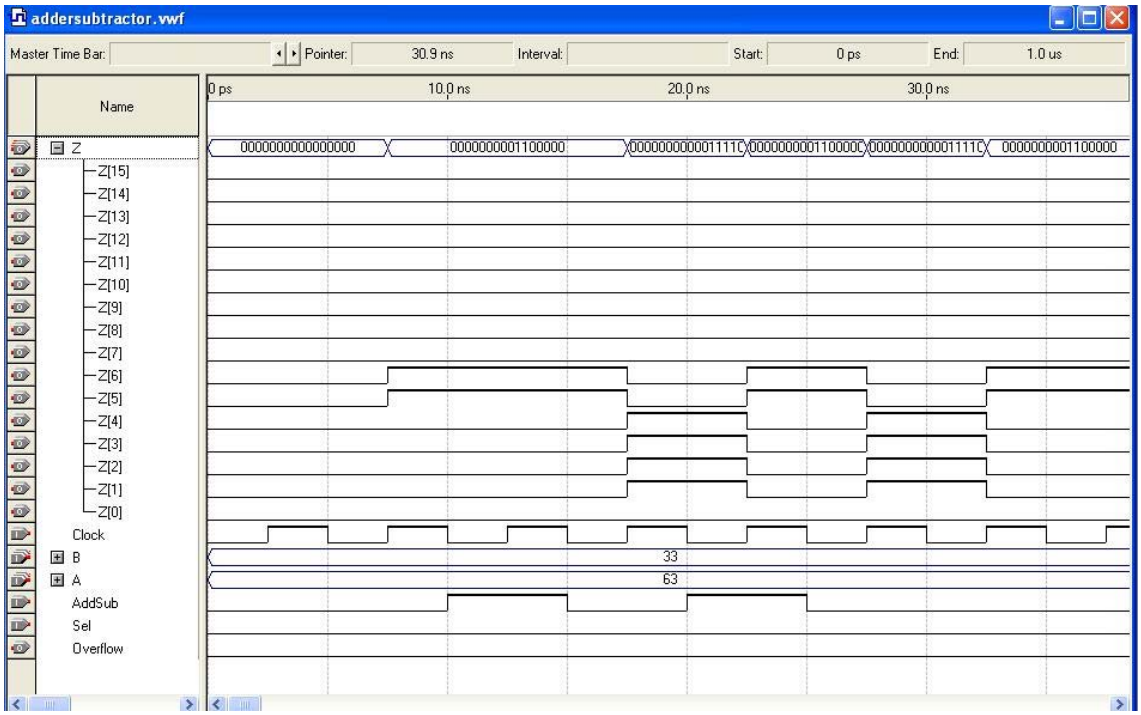
Kuva 21. Vähennys- ja yhteenlaskua toteuttavan kytkennän lohkokaavio [39]

Suunnitelmassa on kaksi varsinaista 16-bittistä sisääntuloa, *A* ja *B*, ja yksi 16-bittinen ulostulo, *Z*. Esimerkissä käytetään 16-bittistä piiriä, jolloin *n* = 16. Sisääntulot ladataan 16-bittisiin rekistereihin *Areg* ja *Breg*. Laskutoimituksen tulos ladataan rekisteriin *Zreg*, josta lähtee piirin ulostulo.

Piirissä on varsinaisten sisääntulojen lisäksi kaksi ohjaustuloa (*AddSub* ja *Sel*). Sisääntulona oleva *AddSub* on ohjaussignaali, joka saa aikaan ulostuloon Z tulojen *A* ja *B* summan (kun *AddSub* = 0) tai erotuksen (kun *AddSub* = 1). Toinen ohjaussisääntulo on *Sel*. Sen ollessa tilassa ”0” suoritetaan yhteenlasku- tai vähennysoperaatio. Tulon ollessa ”1” sisääntulo *B* joko summataan tai vähennetään senhetkisestä ulostulon *Z* arvosta. Jos lisäys- tai vähennystoiminta aiheuttaa ylivuotoa, otetaan käyttöön ulostulosignaali *Overflow*. [39.]

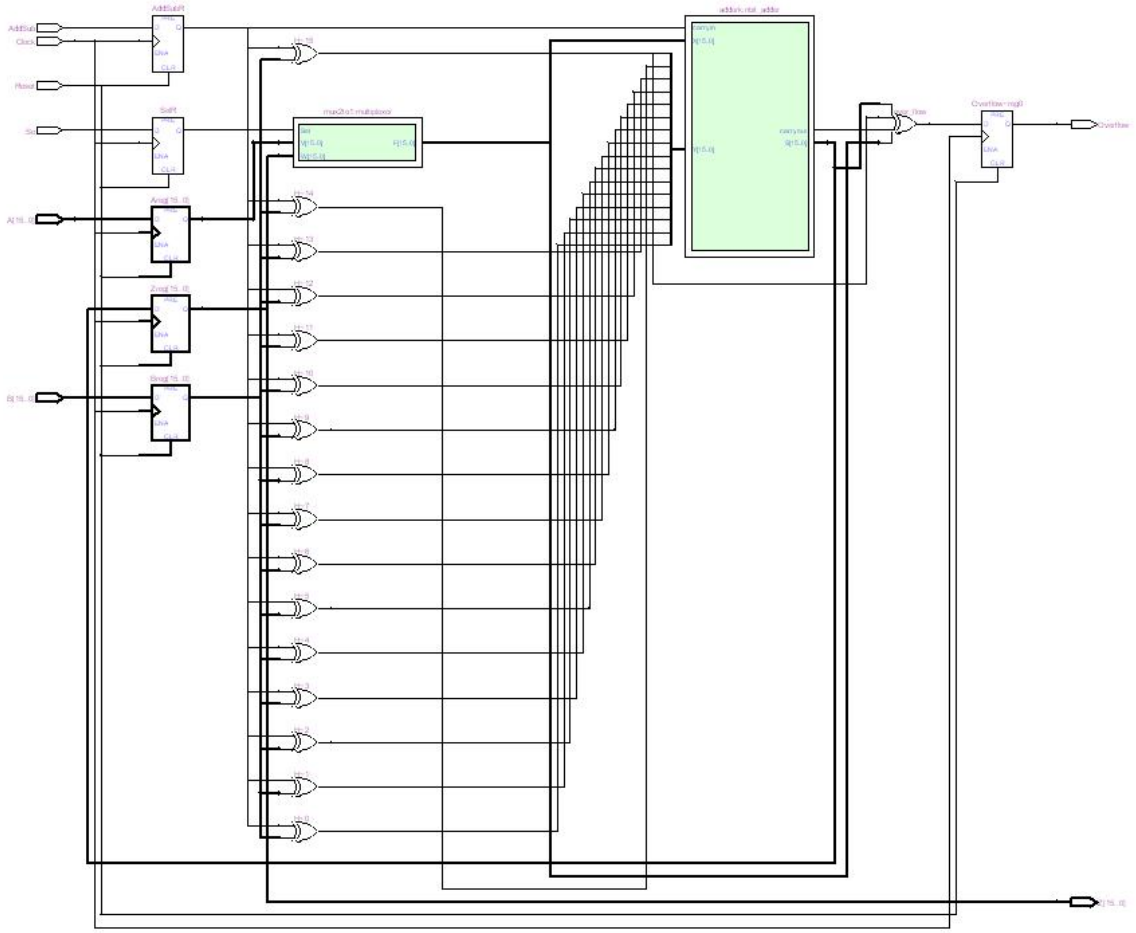
Jotta asynkronisia sisääntuloja olisi helpompi käsitellä, ne ladataan kiikkuihin (flip-flop) kellon positiivisella reunalla. Siten tulot *A* ja *B* ladataan rekistereihin *Areg* ja *Breg*, kun taas *Sel* ja *AddSub* ladataan kiikkuihin *SelR* ja *AddSubR*. [39.]

Suunnittelulle tehtiin synteesi. Synteesin jälkeen suoritettiin simulaatiota Quartus II -ohjelmistoon sisältyvällä simulaatiotyökalulla. Ennen simulaatiota oli generoitava funktionaalisen simulaation kytkentälista ja vwf-tiedosto. Node finder -toiminnon avulla määriteltiin halutut sisääntulot ja ulostulot. Vector Waveform -tiedosto näytti haluttujen kohtien tilat ja esimerkissä sen avulla asetettiin sisään tulevaa dataa eri tiloihin, jolloin nähtiin muutoksia ulostuloissa. Kuvassa 22 on esitetty simulaation tilakaavio. *Addsub*-ohjaustulon muutos saa aikaan yhteen- tai vähennyslaskutoimituksen tulojen *A* ja *B* välillä.



Kuva 22. Vähennys- ja yhteenlaskua toteuttavan piirin suunnittelun simulointi Quartus II -ohjelmiston simulointityökalulla. Lähtörekisterin Z arvo muuttuu ohjaustulojen (AddSub ja Sel) tilojen mukaan.

Tässä vaiheessa ei ollut mahdollista tutkia suunnittelun ajoitustietoja vaan pelkästään toiminnallisuutta. Suunnittelua tutkittiin myös RTL-Viewer -työkalun avulla (kuva 23). Suunnitelma oli nähtävissä rekisterinsiirtotasoisena ja näkymä oli apuna piirin toiminnan ymmärtämisessä. Technology Map Wiever -työkalu tarjosi näkymät suunnittelun eri hierarkiatasoilla.



Kuva 23. Ohjelmiston Quartus II RTL-Viewer -työkalun muodostama kytkentäkuva esimerkkisuunnittelusta

Suunnittelun piiriksi oli valittu Stratix II -piiriperheen FPGA. Sijoittelun ja reitityksen jälkeen oli nähtävissä, että esimerkiksi piirin mukautuvista hakutaulukkoelementeistä oli käytössä alle 1 % (52 kpl) ja pinneistä 15 % (53).

Piirille tehdyn sijoituksen ja reitityksen jälkeen voitiin suorittaa ajoitussimulaatiota sekä ajoitusanalyysi. Analyysi näytti maksimitaajuuden valitulle piirille tehdylle toteutukselle. Arvo on riippuvainen pisimmästä viiveestä kahden rekisterin välillä. Esimerkissä nähtiin, että maksimitaajuutta rajoittava polku alkoi rekisterin *Zreg* bitistä 8 ja päättyi kiikkuun *Overflow*. Timing Closure Floorplan -näkymästä voitiin tarkastella tilannetta piirille tehdyssä sijoituksessa. Jos haluttiin suurentaa taajuutta, voitiin kellon asetusta muuttaa. Ajoitusanalyysi osoitti, että aikarikkomuksia esiintyi. Sijoittelu- ja reititystoiminnon jälkeen piiri oli sijoiteltu ja reititetty uudelleen vastaamaan haluttua nopeutta. Tällöin oli huomatta-

vissa, että valittujen elementtien paikat tai määrät piirillä olivat muuttuneet. Tehty ajoitusanalyysi osoitti, että aikarikkomuksia ei enää ollut.

Jos vaadittu maksimitaajuus asetettiin liian korkeaksi, ei sovitusta piirille voitu tehdä enää virheettömästi. Ajoitusanalyysin tuottamassa raportissa oli nähtävissä, että epäonnistuneita datapolkuja esiintyi eikä aikavaatimuksissa mitenkään voitu pysyä. Suunnittelua oli muutettava siten, että saatiin tyydyttävä suorituskyky ilman aikarikkomuksia.

Ajoitusanalyysi näytti myös muita ajoitustietoja. Kellosta ulostuloon -viive oli piirillä pahimmassa tapauksessa 5,569 ns. Signaalilta rekisteristä *Zreg* (bitti 14) kului em. aika edetä ulostulonastaan *Z* (bitti 14). Muita parametreja olivat asetus- ja pitoaika. Ajoitusvaatimuksia muutettiin esim. kellosta ulostulonastaan ja uudessa piirisijoittelussa oli jälleen nähtävissä muutoksia. Kulloisenkin ajoitusvaatimuksen muuttaminen vaati aina uuden sovituksen piirille. Koska sijoittelu ja reititys on hidas toimenpide, on ajoituksen optimoiminen piirille aikaa vievää toimintaa. Pienessäkin suunnitelmassa se vie helposti useita minuutteja. Myös piirin käytettävä ala muuttuu ajoitusvaatimuksien myötä ja se on otettava suunnittelussa huomioon.

Sijoittelu- ja reititystoiminnon jälkeen oli myös mahdollista editoida suunnitelmaa Chip Editor -työkalun avulla. Suunnittelu oli sijoitettu ja reititetty siten, että tarvittavat nastat ja LAB-elementit (kymmenen logiikkaelementin ryhmä) oli ryhmitelty lähekkäin piirin reunaan. Työkalun avulla muutettiin yhden logiikkaelementtiryhmän paikkaa, tarkastettiin ja talletettiin muutokset kytkentälistaan ja saatiin aikaan uusi sijoittelu ja reititys. Logiikkaelementtiryhmän paikka oli vaihtunut piirin toiseen reunaan, mutta nastojen paikat olivat entisellään. Jos suoritettiin muutos, jonka mukaista sijoitusta ei voitu toteuttaa, palautui edellinen onnistunut sijoittelu voimaan. Täyttä käännöstapahtumaa ei tarvittu. Jos kuitenkin suoritettiin käännös, poistui tehty muunnos ja piiri oli sijoitettu ja reititetty uudelleen. Seuraavaksi voitiin suunnittelu tarkistaa ajoitusanalyysin ja simulaation avulla.

ModelSim-Altera-ohjelman avulla suoritettiin porttitason simulointia. Tällöin käytettiin Quartus II -ohjelmiston generoimaa vhd-tiedostoa. Kytkentälistan ge-

nerointi voitiin suorittaa kun oli määritelty käytettävä EDA-työkalu (ModelSim Altera). Simulointi ModelSim-Altera -työkalun avulla oli monipuolisempaa kuin käytettäessä Quartus II -ohjelmiston omaa simulointiohjelmaa.

Signal Tap II Logic Analyzer -työkaluun ja sen käytettävyyteen etäyhteyden kautta tutustuttiin käyttäen valmista Quartus II -ohjelmiston mukana tulevaa esimerkkisuunnitelmaa (standard.bdf). Suunnitelma, jossa oli mukana myös Nios II -prosessori, oli tehty kokeiltavaksi Stratix II -kokeilulevyn (EP2S60) kanssa. Suunnitelma oli laaja, eikä syvällistä ymmärrystä sen toiminnasta ollut, joten analysaattorin käyttöön tutustuttiin vain pinnallisesti.

Ensin luotiin ja konfiguroitiin tarvittava Signal Tap II -tiedosto (.stp). Tiedosto sisältää asetukset sekä kerätyn datan tutkimista ja analysointia varten. Tiedostoon sisällytettiin tutkittaviksi signaaleja suunnittelusta Node Finder -työkalun avulla. Kelloksi määriteltiin globaali kello. Mikä tahansa signaali olisi ollut mahdollista valita kelloksi, Altera suosittelee kuitenkin globaalin kelloa käyttöä. Kello ohjaa datan keruuta. Näytteistysmääräksi (jokaiselle signaalille tallennettu näytemäärä) määriteltiin 128. Analysaattorin liipaisutyypiksi valittiin ”Don´t Care”. Konfiguroinnin jälkeen tiedosto käännettiin Quartus II -projektin kanssa.

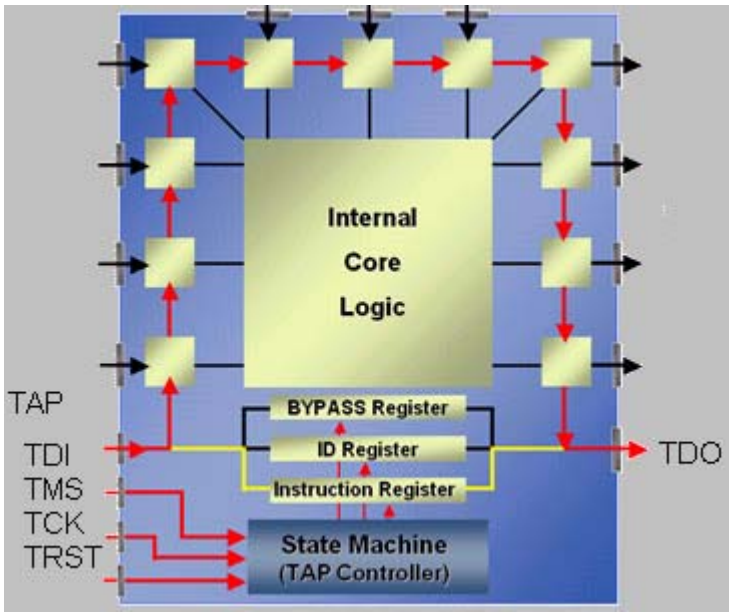
Kokeilulevy EP2S60 oli yhdistetty etätietokoneeseen USB Blaster -ohjelmointi-laitteiston avulla. Tietokoneessa oli asennettuna Quartus II -ohjelmisto. Quartus II Programmer -ohjelmassa konfiguroitiin ohjelmointilaitteisto lisäämällä JTAG-asetuksiin etäkäyttömahdollisuus ja asettamalla salasana etäkäyttäjän pääsyä varten. Paikallisen tietokoneen laitteistoasetuksista muodostettiin yhteys etä-koneelle. IP-osoitteen ja salasanan avulla valittiin etäkoneella käytetty ohjelmointilaitteisto käyttöön. Stp-tiedostoon konfiguroitiin käytettävä etäyhteyden ohjelmointilaite sekä kohteena oleva kokeilulevy ja FPGA:n ohjelmointi suoritettiin sof-tiedoston avulla. Ohjelmoinnin jälkeen voitiin etäkoneeseen liitetylle kokeilulevylle ohjelmoitua suunnitelmaa analysoida normaaleissa käyttöolosuhteissa ja normaalilla nopeudella Signal Tap II -logiikka-analysaattorin avulla.

Saatua dataa voitiin analysoida. Jos haluttiin muuttaa tutkittavia signaaleja tai muita analysaattorin asetuksia, täytyi sekä käännös että ohjelmointi suorittaa uudelleen. (Työkalun Incremental Routing -ominaisuuden avulla muutoksia olisi voinut tehdä tarvitsematta suorittaa täyttä käännöstapahtumaa.) Suunnitelmaan oli mahdollista tehdä korjauksia esimerkiksi Chip Editor -työkalun avulla ja suorittaa uudelleenohjelmointi.

4.4 IEEE 1149.1 (JTAG) -menetelmä Stratix II -piirille

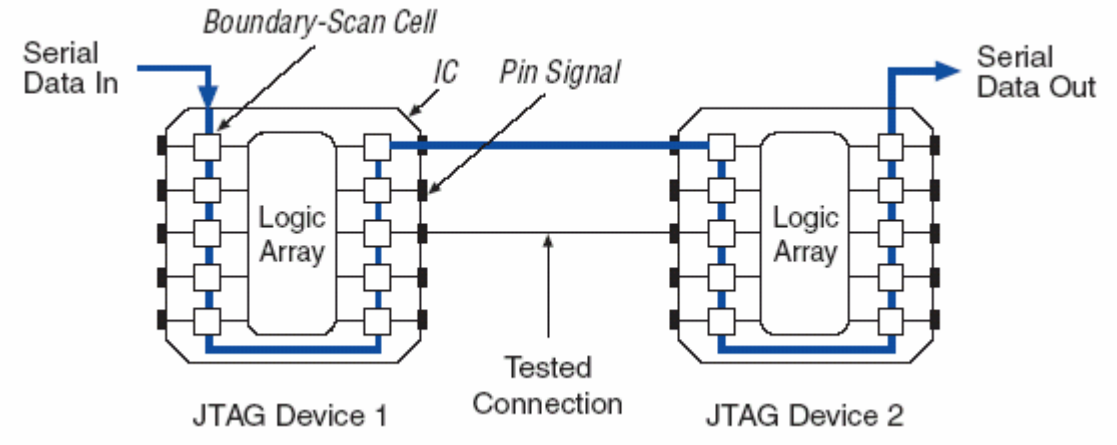
JTAG (Joint Test Action Group) on kehittänyt määrittelyt boundary-scan-testaukselle. Arkkitehtuuri on määritelty standardissa IEEE 1149.1. Standardissa määritellään komponenttien sisäinen testirakenne, komponenttien väliset testiväylät ja testiliityntä. JTAG-menetelmän avulla voidaan testata tiheäjohtimisia komponentteja ja niiden välisiä kytkentöjä piirilevyllä. Fyysisiä mittapäitä ei tarvita ja dataa voidaan kerätä kun laite on normaalisti toiminnassa. Lisäksi JTAG-menetelmän avulla on kortille paikoilleen asennettujen piirien (esim. PLD ja FPGA) ohjelmointi mahdollista. [40.]

Piirissä on sarjaan kytkettyjä boundary-scan-soluja, joiden avulla voidaan asettaa piirin tuloja ja lähtöjä eri tiloihin ja kerätä dataa piirin nastoista. Solu voi olla myös kytkettynä piirin sisäisiin linjoihin. Testidata siirretään sarjamuodossa soluihin. Kerätty data saadaan sarjamuodossa ulos ja sitä voidaan ulkoisesti verrata odotettuun tulokseen. Piirissä on lohko, TAP-ohjain, joka ohjaa JTAG-piiriä sekä piirin tilaa ja testausta. TAP-ohjain on tilakone, jolla on 16 mahdollista tilaa. Standardissa vaaditut rekisterit ovat käskyrekisteri (Instruction Register), ohitusrekisteri (Bypass Register) ja boundary-scan-rekisteri (Boundary-Scan Register). Lisäksi voi olla valinnaisia datarekistereitä, kuten esimerkiksi identifiointirekisteri (ID Register). Käskyrekisteri määrittää käytetyn toiminnon ja datarekisterin. Ohitusrekisteri tarjoaa lyhimmän mahdollisen polun testidatan sisääntulon ja ulostulon välillä. Boundary-scan-rekisteri on siirtorekisteri, joka koostuu piirin boundary-scan-soluista. Kuvassa 24 on esitetty standardin IEEE 1149.1 piiriarkkitehtuuri. [40.]



Kuva 24. Standardin IEEE 1149.1 arkkitehtuuri [40]

Piirilevyllä on usein useampia JTAG-komponentteja. Ne voidaan yhdistää toisiinsa ketjuksi (kuva 25). Toinen vaihtoehto on tehdä useampia ketjuja, jolloin voidaan suorittaa rinnakkaista tarkistusta. [40.]



Kuva 25. Kahden FPGA-piirin muodostama JTAG-ketju [13]

JTAG-menetelmä Stratix II -piirille

Kaikissa Stratix II -piireissä on tuki standardin IEEE 1149.1 mukaiselle boundary-scan-testausmenetelmälle. Testausta voidaan suorittaa ennen tai jäl-

keen konfiguroinnin. JTAG-porttia voidaan käyttää myös piirin ohjelmointiin ja konfigurointiin sekä loogisten toimintojen tarkkailemiseen käytettäessä sulautettua SignalTap II -logiikka-analysaattoria. [41.]

Kun Stratix II -piiri on JTAG-tilassa, käytössä on neljä vaadittavaa nastaa (TDI, TDO, TMS ja TCK) ja yksi valinnainen nasta (TRST). Taulukossa 2 on esitetty nastojen kuvaukset ja toiminnat.

Taulukko 2. Standardin IEEE 1149.1 nastojen kuvaukset [41]

Nasta	Kuvaus	Toiminta
TDI	Test Data input	Sarjamuotoisen datan sisääntulo. Signaalin tila luetaan testikellon nousevalla reunalla.
TDO	Test Data Output	Sarjamuotoisen datan ulostulo. Dataa siirretään ulos kellon laskevalla reunalla.
TMS	Test Mode Select	TAP-ohjaimen (tilakoneen) ohjaussignaali. Tila luetaan kellon nousevalla reunalla.
TCK	Test Clock Input	Synkronointikello sarjaväylälle.
TRST	Test Reset Input (optionaal.)	TAP-ohjaimen asynkroninen reset-signaali. Aktiivinen 0-tilassa.

JTAG-menetelmässä on oleellista, millaista käskykantaa laite tukee. Standardi 1149.1 määrittelee kolme pakollista käskyä, joita ovat EXTEST, SAMPLE/PRELOAD ja BYPASS. Näiden lisäksi Stratix II tukee käskyjä IDCODE, USERCODE, CLAMP ja HIGHZ. [41.]

EXTEST-käskyä käytetään ensisijaisesti testaamaan piirien välisten ulkoisten nastojen yhteydet. SAMPLE/PRELOAD -käskyn avulla voidaan ottaa tilannekuvaus piirin datasta keskeyttämättä sen normaalia toimintaa. Käsky on käytössä myös SignalTap II sulautetussa logiikka-analysaattorissa. BYPASS-käskyn avulla voidaan valitun komponentin sisäinen boundary-scan-ketju ohittaa ja data siirtyy nopeasti seuraavalle piirille. [41.]

IDCODE-käskyn avulla saadaan piirin tunnistetiedot luettua ja USERCODE-käskyllä saadaan esiin laajennettuja tunnistetietoja yms. toiminnan laajennuksia. CLAMP-käsky asettaa piirin lähtönastat haluttuihin tasoihin ja HIGHZ-

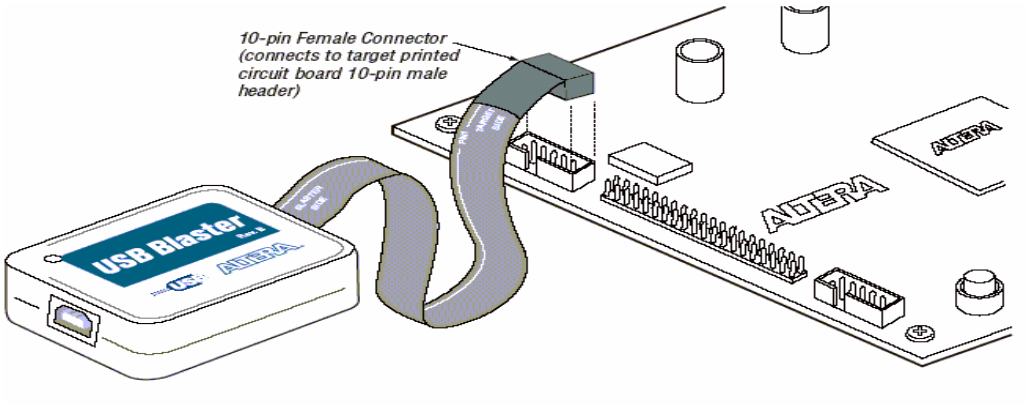
käskyn avulla voidaan piiri erottaa sähköisesti piirilevyltä asettamalla lähdöt kolmitilaan. Lisäksi on käskyjä, joita käytetään konfiguroitaessa Stratix II -piiriä JTAG-portin kautta. [41.]

Käskyrekisterin pituus on 10 bittiä ja Usercode-rekisterin 32 bittiä. Idcode-rekisterin antama tieto vaihtelee Stratix II -piiriperheen piirien välillä, kuten myös boundary-scan-rekisterin pituus. [41.]

Konfigurointi

Stratix II -piirit käyttävät SRAM-soluja konfigurointidatan säilyttämiseen. Koska SRAM on muisti, jonka sisältö katoaa virransyötön loppuessa, on konfigurointi-data ladattava piiriin joka kerta, kun piiriin laitetaan virta. Stratix II -piirit voidaan konfiguroida käyttäen JTAG-järjestelmää. Piirille sisäänrakennettua JTAG-piiristöä käytetään konfigurointidatan siirtämiseen. [42.]

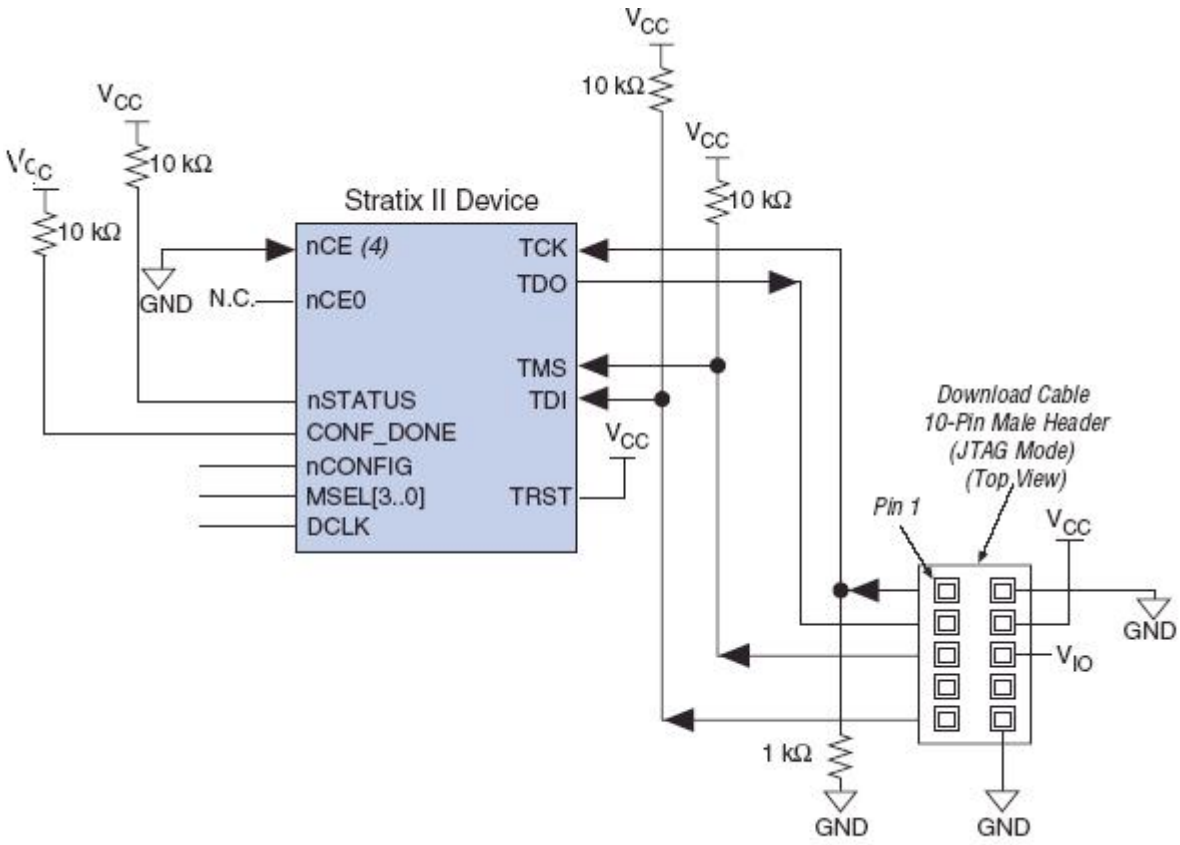
Quartus II -ohjelmisto luo automaattisesti binaarisen SOF-tiedoston (SRAM Object File) kääntäjän Assembler-moduulissa. Tiedosto sisältää SRAM-pohjaisten Alteran piirien konfigurointidataa. SOF-tiedostoa käytetään JTAG-konfigurointiin. Data ladataan Quartus II -ohjelmistoa käyttävältä tietokoneelta kortilla olevaan piiriin käyttämällä USB Blaster-, MasterBlaster-, ByteBlaster II- tai ByteBlasterMV-latauskaapelia (kuva 26). [42.]



Kuva 26. USB Blaster -latauskaapelin avulla voidaan siirtää konfigurointidataa PC:ltä piirille. Kaapeli liitetään PC:n USB-porttiin. [43]

Konfigurointi on samanlaista kuin piirin ohjelmointi, mutta TRST-nasta on yhdistettävä käyttöjännitteeseen. Käytössä ovat lisäksi nastat TDI, TDO, TMS ja TCK. [42.]

Konfiguroitaessa yhtä JTAG-ketjun piiriä, ohjelmointiohjelmisto asettaa kaikki muut piirit BYPASS-muotoon (piiri päästää ohjelmointidatan läpi siten, ettei se kosketa sen sisäistä toimintaa) ja kohteena oleva yksittäinen piiri voidaan ohjelmoida. Sarjamuotoinen konfigurointidata, joka on viety piirin sisään, ilmestyy sen TDO-nastaan yhden kellojakson kuluttua. Quartus-ohjelmisto tarkistaa, että tapahtuma onnistui tutkimalla CONF_DONE-nastan tilaa JTAG-liitännän kautta. Kuvassa 27 on esitetty yhden piirin JTAG-konfigurointi. [42.]



Kuva 27. Piirin JTAG-konfigurointi. Kortilla olevaan liittimeen yhdistetään latauskaapeli (esim. USB Blaster), joka yhdistää kortin tietokoneeseen ja suoritettavaan ohjelmistoon. [42]

Stratix II -piiri on suunniteltu siten, että JTAG-käskyillä on etuoikeus muihin konfigurointimuotoihin. Jos jokin konfigurointimuoto on käynnissä ja JTAG-konfigurointi aloitetaan, keskeytyy edellinen toiminto ja JTAG-konfigurointi alkaa. [42.]

5 PÄÄTELMIÄ VERIFIOINNIN TOTEUTTAMISESTA

SoC-suunnitelman verifiointi on vaativa tehtävä. Suunnitelman olisi oltava mahdollisimman virheetön valmistuessaan eikä sen varmentamiseen saisi mennä kohtuuttoman paljon aikaa. SoC-suunnitelman verifiointi vie yli puolet kokonaissuunnitteluun kuluvasta ajasta. Laadittu verifiointisuunnitelma ja valitut strategiat, työkalut ja käyttöympäristö helpottavat verifioinnin suorittamisessa ja tuotteen saamisessa markkinoille ajoissa. Vaikeinta on päättää, milloin verifiointi riittää. Verifioinnin avulla ei voida osoittaa kaikkia virheitä muutoin kuin käyttämällä virheenhakuun erittäin paljon aikaa. Käytännössä se on harvoin mahdollista. Onkin pystyttävä valitsemaan ja päättämään, mitkä ominaisuudet on saatava verifioitua kattavasti ja mitkä on jätettävä vähemmälle.

Verifiointia suoritetaan suunnittelun jokaisella abstraktiotasolla. Samaa toiminnallisuutta todennetaan siten useita kertoja. Mitä alemmalle abstraktiotasolle verifioinnissa mennään, sitä hitaampaa voi toiminnan varmentaminen olla. Toisaalta abstraktiotason nosto verifioinnin yhteydessä voi nopeuttaa suunnitteluaikaa. On tarpeellista pohtia, missä laajuudessa milläkin tasolla suoritetaan verifiointia.

IP-lohkojen käyttö on runsasta Soc-suunnittelussa. Jos lohkot ovat ennalta verifioituja, voi suunnittelun verifiointiin käytettävä aika lyhentyä. IP-lohkon käyttö uudessa suunnitteluympäristössä aiheuttaa kuitenkin verifioimistarvetta lohkon liittämisessä muuhun suunnitteluun. Huolellisesti ennalta verifioitujen lohkojen käyttö voinee kuitenkin estää pahimmat suunnitteluvirheet.

Simulaation nopeus vaikuttaa suuresti kokonaisverifiointiaikaan. Simulaationopeuteen vaikuttavat suunnittelun koko, käytetyt työkalut ja ympäristö. Porttitasolla suoritettu simulaatio on hidasta nopeankin tietokoneen avulla. Jos suunnitelma on lisäksi suuri eikä työkalukaan ole paras mahdollinen, pitenee simulointiin käytetty aika huomattavasti. Eniten käytetty simulointimenetelmä on tapahtumapohjainen simulaatio. Tekniikka onkin käytössä useimmissa simulaatio-ohjelmistoissa. Vähemmän käytetty tekniikka on jaksopohjainen simulaatio. Se olisi huomattavasti nopeampi, mutta rajoituksena on vaatimus

synkronisesta suunnitelmasta. Simulaation käyttö alkaa tuottaa yhä enemmän ongelmia, kun suunnitelmat kasvavat suuremmiksi. Kaikkien mahdollisten tilojen simulointi suuressa suunnitelmassa alkaa olla yhä vaikeampaa. Uusia ratkaisuja joudutaan varmasti tulevaisuudessa ottamaan käyttöön. Formaali verifiointi saattaa olla eräs mahdollisuus käytettäväksi simuloinnin rinnalla.

Todellinen käyttötilanne on ainoa tapa saada luotettava kuva piirin käyttäytymisestä ja ajoituksista. FPGA:n käytössä on etu verrattuna perinteisiin ASIC-suunnitelmiin, sillä FPGA:n suunnittelussa on mahdollista varhaisessa vaiheessa nähdä piirin toiminta reaalitilanteessa. Piirin uudelleenkonfigurointi voidaan suorittaa, jos ohjelmoinnin jälkeen havaitaan ajoituksellisia tai toiminnallisia virhetilanteita. Perinteisessä ASIC-suunnittelussa käytetyt verifiointimenetelmät siirtyvät usein sellaisenaan käyttöön ryhdyttäessä verifioimaan FPGA-pohjaisia SoC-piirejä. FPGA:n uudelleenohjelmoitavuus on seikka, jonka takia verifiointistrategia on harkittava uudelleen. Tarpeen onkin pohtia, mitkä seikat on parasta varmentaa ennen piirin ohjelmointia ja mitkä taas korttitasolla piirin varsinaisen ohjelmoinnin jälkeen. Vaikka FPGA-piiriä käytettäessä on mahdollista myöhäisessäkin vaiheessa korjata virhe suunnitelmassa, on suunnitelmaa silti verifioitava tarpeellisessa määrin ennen ohjelmointia.

FPGA:n suunnitteluaikaa ei tarvitse käyttää itse piirin testaukseen, sillä valmistaja on testannut piirin ennen sen toimitusta. Voidaankin keskittyä simulaatioon ja muuhun suunnitelman verifiointiin.

Alteran omassa FPGA:n Quartus II -suunnitteluohjelmistossa on erilaisia verifiointityökaluja. Ennen implementaatiota käytettäviä työkaluja ovat simulaattori ja ajoitusanalyysi. Ohjelmiston oma simulaatio-ohjelma on vaatimaton tapahtumapohjainen työkalu. Suositeltavaa onkin käyttää ohjelmiston lisenssin mukana tulevaa Quartus II -käyttäjille tehtyä ModelSimAltera-simulointityökalua. Muitakin vaihtoehtoja on tarjolla. Ajoitusanalyysiä varten on Quartus II -ohjelmistossakin nykyisin yleinen staattinen ajoitusanalyysi -työkalu. Analyysiä voidaan kuitenkin hyödyntää vain synkronisissa suunnitelmissa. Tehon arviointiin ja analysointiin löytyvät myös Quartus II -ohjelmistosta työkalut, mutta työssä ei perehdytty niiden käyttöön.

Ohjelmoinnin jälkeen voidaan korttitason testauksessa käyttää Quartus II -ohjelmiston sulautettua logiikka-analysaattoria, johon työssä tutustuttiin. Se antaa tietoja loogisista toiminnoista JTAG-portin kautta. Työkalun tehokkaaseen käyttöön vaaditaan kuitenkin syvällistä ymmärrystä suunnitelmasta ja tietoa siitä, miten suunnitelman todella pitäisi toimia. Tarvittaisiinkin pitempiaikaista perehtymistä, jotta työkalun ominaisuuksia voitaisiin hyödyntää tai tarkastella kriittisesti. Työkalun vertailu laitteistopohjaiseen logiikka-analysaattoriin antaisi myös tietoa sen ominaisuuksista.

6 YHTEENVETO

Verifioinnin avulla varmennetaan määrittelyjen mukaisen SoC:n suunnittelun virheettömyys ja sen avulla vastataan suunnittelun laadusta. Verifiointia käytetään FPGA:n suunnitteluprosessin eri vaiheissa ennen implementointia sekä implementoinnin jälkeen. Työssä perehdyttiin eri verifiointimenetelmiin sekä Alteran oman Quartus II -suunnitteluohjelmiston verifiointimenetelmiin. Työ oli osa projektia, jolla haluttiin lisätä SoC-tietämystä. Tätä tietämystä voivat alan yritykset hyödyntää toiminnassaan.

Verifiointitekniikoita on useita. Ne voidaan jakaa karkeasti neljään luokkaan: simulointipohjaiset tekniikat, staattiset tekniikat, formaaliset tekniikat sekä fyysinen verifiointi ja analyysi. SoC:n suunnittelussa suunnittelun toiminnallinen verifiointi, simulointi, vie suuren osan suunnitteluajasta ja sitä käytetään useassa vaiheessa suunnittelua ajoituksen ja toiminnan verifiointiin. Simulointitapoja ja -tekniikoita on useita (tapahtumapohjainen, jaksopohjainen, emulointi, nopeat prototyypit, laitteistokiihdyttimet jne.) Toinen paljon käytetty verifiointimenetelmä FPGA:n suunnittelussa on staattinen ajoitusanalyysi. Ajoituksen analysoinnin merkitys onkin koko ajan kasvanut. Eri verifiointitekniikat tarjoavat erilaisia etuja ja ominaisuuksia ja erilaisille suunnitelmille on löydettävä parhaat verifiointimenetelmät.

Vaikka suunnitelmaa olisi simuloitu kuinka paljon tahansa, on piirikorttitasolla tapahtuva testaus välttämätöntä. Jokaisella suunnitelmalla on omat ajoitusmuunnelmansa ja sähköinen ympäristönsä. Sisäisiä signaaleja voidaan tutkia esim. logiikka-analysaattorin avulla.

Eräs suuri FPGA:n etu ASIC-piireihin nähden on sen uudelleenohjelmoitavuus. Ominaisuus vähentää myös simuloinnin kuormitusta suunnittelussa.

Alteran oma suunnitteluohjelmisto tarjoaa verifiointimenetelmiä eri suunnitteluvaiheisiin. Simulointia voidaan suorittaa eri suunnitteluvaiheissa ohjelmiston omalla tai lisenssin mukana saatavalla ModelSim Altera -simulointityökalulla. Quartus II -ohjelmistossa on työkalu staattisen ajoitusanalyysin suorittamiseen.

Tehonkulutuksen arviointiin on myös työkaluja. Implementoinnin jälkeistä in-system-testausta voidaan suorittaa sulautetun logiikka-analysaattorin (SignalTap II Logic Analyzer) avulla tai käyttämällä SignalProbe-työkalua. Työssä on tutkittu ja testattu Quartus II -ohjelmiston verifiointityökaluja ja niiden käyttöä esimerkkisuunnitelman avulla.

LÄHDELUETTELO

1 Rashinkar, P., Paterson, P. & Singh, L. System-on-a-Chip Verification - Methodology and Techniques. Boston: Kluwer, 2001. 372 s. ISBN 0-7923-7279-4.

2 Chien-Nan, L. SoC Verification Methodology. [WWW-dokumentti] <http://www.cs.ccu.edu.tw/~pahsiung/courses/soc/notes/04_Verify.pdf>. (Luettu 12.12.2004.)

3 Teixeira, J. Design and Test of Electronic Systems.2003. [WWW-dokumentti] <http://figaro.inesc-id.pt/dft/DfT-1.pdf>. (Luettu 1.2.2005.)

4 Nardi, A. Digital Systems Verification. [WWW-dokumentti] <http://www-cad.eecs.berkeley.edu/~nardi/EE219A/lectures/lec13_bw_2xp.pdf>. (Luettu 2.2.2005.)

5 Bergeron, J. Writing Test Benches - Functional Verification of HDL Models, Boston, Mass.: Kluwer, 2000.

6 Chien-Nan, L. Introduction to SoC Design and Verification Flows [WWW-dokumentti] <http://www.ee.ncu.edu.tw/~jimmy/courses/DVM04/01_intro.pdf >. (Luettu 12.12.2004.)

7 Ziv, A. Functional Verification and the SoC Challenge. 2003. IBM Research Lab in Haifa.IBM Corporation. [WWW-dokumentti] <http://tima.imag.fr/MPSOC/2003/slides/soc_verification.pdf>. (Luettu 3.2.2005.)

8 Pietikäinen, V. Järjestelmäsirut. Nokia, 2001. [WWW-dokumentti] <http://tisu.mit.jyu.fi/embedded/TIE345/vanhat_luentokalvot/soc_10.pdf>. (Luettu 3.3.2005.)

9 Nekoogar, F. & Nekoogar, F. From ASICs to SOCs, A Practical approach. Pearson Education, Inc. 2003. 192 s. ISBN 0-13-033857-5

10 Mosensoson, G. Practical Approaches to SOC Verification. White Paper. Verisity Ltd. Julkaistu 25.7.2001. [WWW-dokumentti] <http://www.verisity.com/resources/whitepaper/technical_paper.html>

11 Haltsonen, S., Levomäki, J. & Rautanen, E. Digitaalitekniikka. 1. painos. Helsinki: Edita Prima Oy, 2004. 400 s. ISBN 951-37-3886-8.

12 Kalifornian yliopisto, Berkeley. Lab 1 FPGA CAD Tools. 2005. [WWW-dokumentti] <http://www-inst.eecs.berkeley.edu/~cs150/sp05/Lab/Lab1.PDF>. (Luettu 12.3.2005.)

13 Altera Corporation: Internet-sivustot. < http://www.altera.com>

14 Tessier, T. Rethinking Your Verification Strategies for Multimillion-Gate FPGAs, versio 1.2. Julkaistu 15.2.2002. [WWW-dokumentti] <http://direct.xilinx.com/bvdocs/appnotes/xapp408.pdf>

15 Olukotun, K., Heinrich, M. & Ofelt, D. Digital System Simulation: Methodologies and Examples. Stanfordin yliopisto, Computer Systems Laboratory. 1998. [WWW-dokumentti] <http://www.cs.ucr.edu/~dalton/refdesign/docs/ stanford_digsim_dac98.pdf>. (Luettu 22.2.2005.)

16 Cyber Caist. CAD issues & Algorithms (1). [WWW-dokumentti] <http://vswww.kaist.ac.kr/source/course_work/course/ee878-2002/ lecture/SoC_18-CAD1-1030.pdf>. (Luettu 14.1.2005.)

17 Wolf, W. Modern VLSI Design, System-on-Chip Design, Chapter 8. 1998, 2002. [WWW-dokumentti] <http://www.princeton.edu/~wolf/modern-vlsi/ Overheads.html>. (Luettu 31.3.2005)

18 Eder, K. Verification. Bristolin yliopisto. Julkaistu 13.10.2004. [WWW-dokumentti] <http://www.cs.bris.ac.uk/Teaching/Resources/ COMSM0115/2_Verification_Tools.v.pdf>

19 Culler, D. EECS 150 - Components and Design Techniques for Digital Systems Lec 04 - Hardware Description Languages/ Verilog. Kalifornian yliopisto, Berkeley. 2004. [WWW-dokumentti] <http://www-inst.eecs.berkeley.edu/~cs150/fa04/Calendar.htm>. (Luettu 5.11.2004.)

20 Cadence: Internet-sivustot. <http://www.cadence.com>

21 Higashi, A., Tamaki, K. & Sasaki, T. Verification Methodology for a Complex System-on-a-Chip. 1999 [WWW-dokumentti] <http://magazine.fujitsu.com/us/vol36-1/paper05.pdf>. (Luettu 10.3.2005.)

22 Sander, I. Hardware Accelerators. [WWW-dokumentti] <http://www.imit.kth.se/courses/2B1447/0304/Accelerators.pdf>. (Luettu 10.3.2005.)

23 Wikipedia Encyclopedia. Tietosanakirja. <http://en.wikipedia.org/wiki >. (Luettu 1.2.2005.)

24 Negookar, F. Timing Verification of Application-Specific Integrated Circuits (ASICs). Prentice Hall PTR. 1999. ISBN 0-13-794348-2

25 Altera Corporation. Literature: Timing Analysis. Päivitetty 1/2005. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii5v3_02.pdf>

26 Synopsis: Internet-sivustot. <http://www.synopsys.com>

27 McCarty, D. Deliver Products On-Time with RTL Hardware Debug. Julkaistu 11.1.2005. Fpga and Programmable Logic Journal Internet-sivustot. [WWW-dokumentti] <http://www.fpgajournal.com/articles_2005/20050111_synplicity.htm>. (Luettu 12.3.2005.)

28 Altera Corporation. Quartus II -ohjelmisto: tutorial_quartusii_intro_vhdl.pdf

29 Altera Corporation. Quartus II -ohjelmisto: Tutorial: Compiler.

30 Altera Corporation. Literature: Simulation. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii5v3_01.pdf>

31 Altera Corporation. Literature: Timing Analysis. Päivitetty 1/2005. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53004.pdf>

32 Altera Corporation. Literature: Power Estimation & Analysis. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii5v3_03.pdf>

33 Altera Corporation. Literature: On-Chip Debugging. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii5v3_04.pdf>

34 Altera Corporation. Literature: Quick Design Debugging Using SignalProbe. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53008.pdf>

35 Altera Corporation. Literature: Design Debugging Using the SignalTap II Embedded Logic Analyzer. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53009.pdf>

36 Altera Corporation. Literature: Design analysis & Engineering Change Management with Chip Editor. Päivitetty 1/2005. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53010.pdf>

37 Altera Corporation. Literature: In-System Updating of Memory & Constants. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53012.pdf>

38 Altera Corporation. Literature: Cadence Incisive Conformal Equivalency Checker Support. Päivitetty 12/2004. [WWW-dokumentti] <http://www.altera.com/literature/hb/qts/qts_qii53011.pdf>

39 Altera Corporation. Quartus II -ohjelmisto: Quartus II Introduction for Verilog Users.pdf

40 JTAG Technologies: Internet-sivustot. <http://www.jtag.com>

41 Altera Corporation. Literature: Configuration & Testing. Päivitetty 1/2005. [WWW-dokumentti] <http://www.altera.com.cn/literature/hb/stx2/stx2_sii51003.pdf>

42 Altera Corporation. Literature: Configuring Stratix II Devices. Päivitetty 1/2005. [WWW-dokumentti] <http://www.altera.com/literature/hb/cfg/stx2_sii52007.pdf>

43 Altera Corporation. Literature: USB-Blaster Download Cable User Guide. Päivitetty 12/2004. [WWW-dokumentti] <www.altera.com/literature/ug/ug_usb_blstr.pdf>

LIITTEET

```
// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
parameter n = 16;
input [n-1:0] A, B;
input Clock, Reset, Sel, AddSub;
output [n-1:0] Z;
output Overflow;
reg SelR, AddSubR, Overflow;
reg [n-1:0] Areg, Breg, Zreg;
wire [n-1:0] G, H, M, Z;
wire carryout,over_flow;
// Define combinational logic circuit
assign H = Breg ^ {n{AddSubR}};
mux2to1 multiplexer (Areg, Z, SelR, G);
defparam multiplexer.k = n;
adderk nbit_adder (AddSubR, G, H, M, carryout);
defparam nbit_adder.k = n;
assign over_flow = carryout ^ G[n-1]^ H[n-1]^M[n-1];
assign Z = Zreg;
// Define flip-flops and registers
always @(posedge Reset or posedge Clock)
if (Reset == 1)
begin
Areg <= 0; Breg <= 0; Zreg <= 0;
SelR <= 0; AddSubR <= 0; Overflow <= 0;
end
else
begin
Areg <= A; Breg <= B; Zreg <=M;
SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
end
endmodule
// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Sel, F);
```

```
parameter k = 8;
input [k-1:0] V, W;
input Sel;
output [k-1:0] F;
reg [k-1:0] F;
always @(V or W or Sel)
if (Sel == 0) F = V;
else F = W;
endmodule

// k-bit adder
module adderk (carryin, X, Y, S, carryout);
parameter k = 8;
input [k-1:0] X, Y;
input carryin;
output [k-1:0] S;
output carryout;
reg [k-1:0] S;
reg carryout;
always @(X or Y or carryin)
{carryout, S} = X + Y + carryin;
endmodule
```