TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

ELEKTRONIIKAN JA TIETOLIIKENNETEKNIIKAN TUTKINTO-OHJELMA

KANDIDAATINTYÖ

Syntetisoitavien RTL-mallien koodaussäännöt

Tekijä Rasmus Kestilä

Ohjaaja Jukka Lahti

Tammikuu 2023

Kestilä R. (2023) Syntetisoitavien RTL-mallien koodaussäännöt. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 26 s.

TIIVISTELMÄ

Tässä kandidaatin työssä perehdytään RTL-mallien koodaukseen, niin että kirjoitettu koodi voidaan syntetisoida logiikkaporteiksi sekä rekistereiksi.

Työssä käydään aluksi läpi kombinaatio- sekä sekvenssilogiikan perusteet ja ominaisuudet. Myöhemmässä vaiheessa tarkastellaan, millaisia vaatimuksia syntetisoitavilla kombinaatio- ja sekvenssilogiikkamalleilla on, jotta ne toimivat oikein.

Lisäksi työ sisältää esimerkkejä virheellisistä RTL-malleista ja niiden tutkimista. Virheellisiä malleja myös tarkastetaan Cadence JasperGold SuperLint -ohjelmalla ja tarvittavat korjaukset käydään läpi niin, että RTL-mallit pystytään syntetisoimaan.

Avainsanat: Rekisterisiirtotaso, Syntetisointi, RTL-malli, Kombinaatiologiikka, Sekvenssilogiikka, SystemVerilog, HDL.

Kestilä R. (2023) Coding rules for synthesizable RTL models. University of Oulu, Degree Programme in Electronics and Communications Engineering. Bachelor's Thesis, 26 p.

ABSTRACT

In this bachelor's thesis we familiarize ourselves with the coding of RTL models, so that these models can be synthesized to logic gates and registers.

The work initially covers the basics and features of combinational and sequential logic. Later we will look at what kind of requirements does synthesizable combinational and sequential logic models need to work correctly.

In addition, the work includes examples of incorrect RTL models and examining of them. The incorrect models are also checked with Cadence JasperGold SuperLint tool and necessary corrections are looked over, so that the RTL models can be synthesized.

Key words: Register Transfer-Level, Synthesizing, RTL model, Combinational logic, Sequential logic, SystemVerilog, HDL.

SISÄLLYSLUETTELO

ALKULAUSE

Tämä kandidaatin työ oli mielenkiintoinen ja mukava tehdä. Koen, että opin sen avulla paljon uutta, ja se myös palautti mieleen aikaisemmin opeteltuja asioita, jotka osaan nyt entistäkin paremmin. Kiitokset Jukka Lahdelle mielenkiintoisesta aiheesta, tämän työn ohjauksesta sekä digitaalitekniikka kurssien opettamisesta.

Oulussa 17.01.2023

Rasmus Kestilä

LYHENTEIDEN JA MERKKIEN SELITYKSET

IC	Integrated Circuit
RTL	Register Transfer-Level
HDL	Hardware Description Language
MUX	Multiplexer
clk	Kellosignaali
rst_n	Reset-signaali, aktiivinen alhaalla (Active low)
Posedge	Kellon positiivinen reuna
Negedge	Kellon negatiivinen reuna
Lint	Koodin analysointityökalu
=	Estomääritys (Blocking)
<=	Estoton määritys (Nonblocking)

1 JOHDANTO

Digitaalisten IC-piirien suunnittelijoilta pyydetään jatkuvasti vaativampia tuotteita, jotka yhdistelevät yhä enemmän toimintoja yhdelle pienelle sirulle. Kasvavat markkinapaineet rajoittavat tuotekehitykseen kuluvan ajan käyttöä. Tämän lisäksi tuotteiden tuloaikojen markkinoille odotetaan lyhenevän jatkuvasti. Tämän takia tuottavuuden sekä suunnittelun on kehityttävä koko ajan. Onneksi laitteistokuvauskielet eli HDL (engl. Hardware Description Language) sekä suunnittelun automaatio tulevat apuun eri tavoin. HDL-synteesiä käytetään jatkuvasti RTL-koodin muuttamiseksi porttitason esityksiksi, joita käsitellään erilaisten solupohjaisten automaatio ohjelmien avulla. [1]

RTL eli rekisterisiirtotaso (Register Transfer-Level) on matalan tason abstraktio, joka mahdollistaa digitaalisen piirin kuvauksen. RTL voi myös viitata kieleen, jolla kuvataan laitteiston toimintaa. Näitä ovat esimerkiksi VHDL, Verilog sekä SystemVerilog. Rekisterisiirtotaso perustuu synkroniseen logiikkaan, ja sen toiminta perustuu kolmeen tärkeimpään osaan, joihin kuuluvat rekisterit, kombinaatiologiikka, sekä kellot. [2][3]

Digitaaliset piirit esittävät informaation käyttäen vain kahta jännite tasoa. Tämän ansiosta piirien luotettavuus ja tarkkuus parantuvat, sekä se tuo lukuisia muita etuja piirien suunnittelussa. [4]

Asioiden ja kehitelmien mallinnus on todella tärkeä osa suunnittelua. Malleja voidaan ajatella eräänlaisiksi esityksiksi objekteista, jotka sisältävät tiettyyn tarkoitukseen tärkeitä näkökulmia, mutta jättää huomiotta aspekteja, jotka eivät välttämättä liity sillä hetkellä asiaan tai ole merkityksellisiä. Mallit ovat siis objektin abstraktioita. Tätä samaa ajattelua käytetään myös RTL-mallien kanssa. [4]

Digitaalisen piirin suunnittelussa jatkuvien prototyyppien tekeminen, ja niillä testaaminen olisi sekä rahaa että aikaa vievää. Silloin piiriä jouduttaisiin testaamaan uudelleen useita kertoja, jotta se pystyisi suorittamaan vaadittuja toimintoja erilaisten rajoitusten mukaisesti. Paljon tehokkaampi tapa on kehittää malleja suunnitelmasta ja arvioida sekä tutkia niitä. Yksi esimerkki digitaalisen piirin RTL-mallista voisi olla esimerkiksi malli, joka esittää piirin loogista rakennetta eli sitä tapaa, jolla piiri on yhdistetty toisiinsa erilaisilla komponenteilla, kuten porteilla ja kiikuilla. [4]

RTL-tason koodaus täytyy tehdä siten, että mallit voidaan syntetisoida. Tällä tarkoitetaan sitä, että RTL-mallit pystytään muuntamaan logiikkasynteesityökaluja käyttäen porttitason abstraktioksi. Tämän avulla luodaan suunnitelma, joka sisältää portteja ja rekistereitä, jota käytetään kaikkiin myöhemmän vaiheen toteutuksiin. [3]

Tämän kandidaatintyön tavoitteena on perehtyä RTL-malleihin ja niiden käyttämiseen tarvittaviin koodaussääntöihin. Tämä tehdään tutkimalla rekisterisiirtotason mallien vaatimuksia, ja tarkastelemalla virheellisiä malleja lint-työkalun avulla. Lopullisena tavoitteena on varmistua siitä, millaisia RTL-malleja voidaan syntetisoida, ja millaiset saavat aikaan odottamattomia virheitä.

2 TEORIA

RTL-malleilla sekä yleisesti digitaalitekniikalla toteutetuissa malleissa tiedon käsittely perustuu kahteen perusoperaatioon, jotka ovat tiedon varastoiminen sekä tiedon muuntaminen eri muotoon tai eri paikkaan. Tiedon muuntaminen tarkoittaa piirin ulostulojen määrittymistä sisääntulojen eri yhdistelmien eli kombinaatioiden avulla. Tällä tavalla binäärikoodia voidaan muuttaa toiseksi ennalta suunnitellulla tavalla. Tässä kohdassa ei keskitytä tiedon tallentamiseen tai varastoimiseen, eikä piiri muista edellistä tilaansa, vaan tulo riippuu pelkästään eri yhdistelmistä. Tällaista piiriä kutsutaan kombinaatiopiiriksi, joka käyttää hyväkseen kombinaatiologiikkaa. [4]

Vaikka kombinaatiopiirit ovat oleellisena osana digitaalisia piirejä, melkein kaikki digitaaliset systeemit käyttävät hyväkseen sekvenssilogiikkaa. Tämä mahdollistaa tiedon tallentamisen, eli piiri pystyy muistamaan sen edellisen tilansa. Tällöin lähdön eli ulostulon arvot määräytyvät sekä nykyisten että edellisten tuloarvojen perusteella. [4]

2.1 Kombinaatiologiikka

Kaikki digitaaliset logiikkapiirit pitävät sisällään loogisia 1 bittisiä lähtöjä sekä tuloja, joiden arvo voi olla millä hetkellä hyvänsä tosi tai epätosi, eli jännite on joko korkea tai matala. Näitä kahta jännitetasoa ajatellaan sähköisenä Boolen logiikkana, ja ne ovat aina joko 0 tai 1. Kombinaatio piireissä jokainen lähtö on yhden tai useamman tulon Boolen funktio. Tämä tarkoittaa sitä, että jokaiselle tulon arvojen kombinaatiolle on lähdössä Boolen algebran mukainen arvo. Kombinaatiopiirille oleellista on se, että se ei muista edellistä tilaansa, vaan arvot tulevat suoraan eri porttien yhdistelmistä. [4]

Boolen funktioita voidaan määrittää monella tapaa, mutta suorin ja yksinkertaisin keino on listata niiden tuloarvot jokaiselle syötettyjen arvojen yhdistelmälle. Tällaista taulukkoa kutsutaan totuustauluksi. Alla oleva taulukko 1 esittää kolmen yleisen loogisen perusoperaation totuustaulut. Operaatiot ovat TAI, JA sekä negaatio eli looginen EI (engl. OR, AND, NOT). [4]

Loogisen TAI operaation merkki on ”+” eli sen tulo (engl. Output) on x + y, ja sitä kuvaa taulukon 1 sisältämä vasemmanpuoleinen totuustaulu. Sen tarkoitus on määrittää lähtö niin, että joko tulo x on 1 tai tulo y on 1. Loogisen JA operaation merkki on ”*” eli sen tulo on x * y, ja sitä kuvaa taulukon 1 sisältämä keskimmäinen totuustaulu. JA funktion ulostulo kertoo, milloin tulot x ja y ovat samaan aikaan totta eli 1. Looginen negaatio eli NOT operaattori on nimensä mukaisesti negaatio sen sisääntulosta. Yksinkertaisesti lähtö on totta, kun tulo on epätosi, ja toisinpäin. Sen totuustaulu löytyy taulukon 1 oikeasta reunasta.

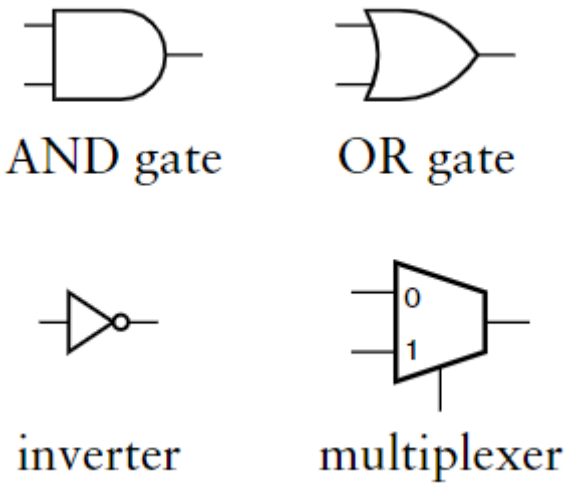
Taulukko 1. Loogisten perusoperaatioiden totuustaulut (OR, AND, NOT)

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A * B
0	0	0
0	1	0
1	0	0
1	1	1

A	$\overline{A}$
0	1
1	0

Käytännössä logiikkaporttien binääriset arvot 0 ja 1 toteutetaan pienjännitteisinä sekä korkeajännitteisinä digitaalisten signaalien arvoina. Tällöin 0 on pieni jännite ja 1 korkea jännite. Jokaiselle logiikkaoperaattorille on omat logiikkaporttinsa, jotka näkyvät alla olevassa kuvassa 1. Samaan kuvaan on myös lisätty neljäs portti nimeltä multiplekseri tai lyhyesti MUX (engl. Multiplexer), joka toimii datan valitsimena. Sen saaman valitsin signaalin avulla päätetään, kummasta portista halutaan tulo, joka laitetaan lähtöön. Multipleksereitä voi myös käyttää useammalla tulosignaalilla, ja valita useita aktivointi signaaleita käyttäen haluttu lähtö. [4]

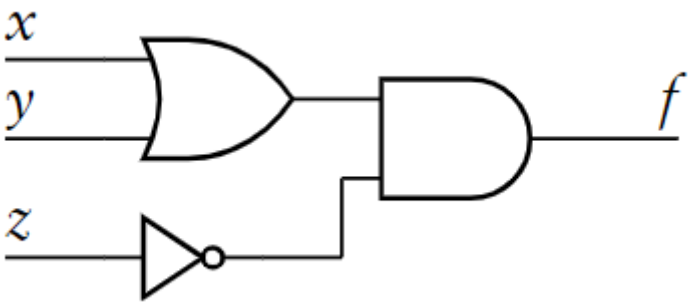


Kuva 1. Digitaaliset logiikkaportit [4].

Logiikkaportteja yhdistelemällä voidaan rakentaa Boolen lausekkeiden mukaisia piirejä. Esimerkiksi yhdistelemällä kolmea edellä mainittua porttia, voidaan toteuttaa seuraava Boolen kaava (1).

$$f = (x + y) * \overline{z} \quad (1)$$

Tämä saadaan muutettua logiikkapiiriksi käyttämällä muuttujille x ja y TAI-porttia, z muuttujalle negaatioporttia, ja yhdistämällä ne JA-portilla alla olevan kuvan 2 mukaisesti.



Kuva 2. Boolen lauseketta (1) kuvaava piiri [4].

Jokaista Boolen lauseketta kohti voidaan kirjoittaa oma totuustaulu, joka sisältää jokaisen lausekkeen muuttujan ja niiden mahdolliset arvot. Boolen lausekkeessa olevien muuttujien lukumäärä määrää totuustaulun rivien lukumäärän alla olevan kaavan (2) mukaisesti, jossa muuttuja n on lausekkeen muuttujien lukumäärä. [4]

$$\text{rivien lukumäärä} = 2^n \tag{2}$$

Kun kuvan 2 logiikkapiiristä halutaan tehdä totuustaulu, täytyy siinä esittää jokainen mahdollinen tulo sekä lähtö. Koska muuttujia x, y ja z on nyt kolme, tarvitaan rivejä kahdeksan kappaletta. Näin saadaan taulukon 2 mukainen totuustaulu.

Taulukko 2. Boolen lausekkeen mukainen totuustaulu

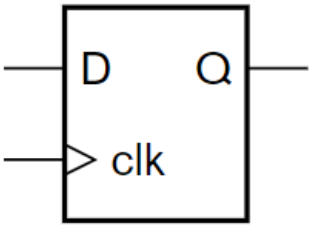
x	**y**	**z**	**(x + y) * z**
0	**0**	**0**	**0**
0	**0**	**1**	**0**
0	**1**	**0**	**1**
0	**1**	**1**	**0**
1	**0**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**1**
1	**1**	**1**	**0**

2.2 Sekvenssilogiikka

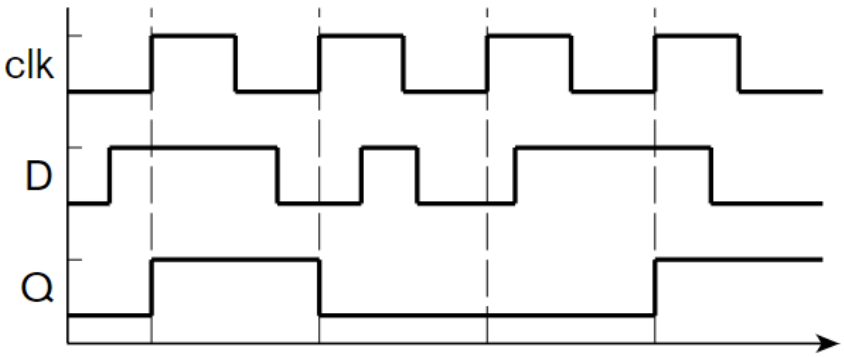
Toisin kuin kombinaatiopiirissä, sekvenssipiirin lähtö ei perustu pelkästään nykyisiin arvoihin, vaan se käyttää hyödykseen myös edellisiä tulon arvoja piirissä. Sekvenssipiirit muistavat edellisiä ajan hetkiä, ja niillä on jonkinlainen varasto tai muisti, johon informaatiota säilötään tietyn ajan. Sekvenssipiirit ja niiden logiikka perustuu vahvasti jaksottaiseen kellosignaaliin, joka jakaa ajan diskreetteihin kellon jaksoihin. Tämän avulla esimerkiksi muistit pystyvät toimimaan esimerkiksi jokaisella positiivisella kellosignaalin nousulla, ja tällä tavoin pitää vanhan tilansa kellon jakson ajan. [4]

2.2.1 D-kiikku

Yksi sekvenssilogiikan yksinkertaisimpia rakennuspalasia ovat D-kiikut (engl. D flip-flop). Ne pystyvät varastoimaan kerrallaan yhden bitin informaatiota, ja ne ovat reuna aktiivisia, eli jokaisella kellon positiivisella reunalla, sen hetkinen informaatio D-kiikun tulossa tallentuu kiikun muistiin. Samaan aikaan muistissa oleva bitti näkyy kiikun lähdössä, jota merkataan kuvassa 3 kirjaimella Q. D-kiikun ajoitusta kuvaava kaavio on esitetty kuvassa 4, ja siitä näkee, kuinka informaatio tallentuu kiikun muistiin eri ajan hetkillä. Ajoituskaavio esittää kiikun ideaalista toimintaa, eli se ei ota huomioon mahdollisia viiveitä. D-kiikuissa on myös usein lisäominaisuuksia, kuten asynkroninen nollaustulo (engl. Asynchronous reset), jolla kiikun tila voidaan pakottaa nollaksi, millä hetkellä hyvänsä. Useasti kiikuissa on myös komplementoitu lähtö, joka muuttaa lähdön Q arvon negaatioksi. [4]

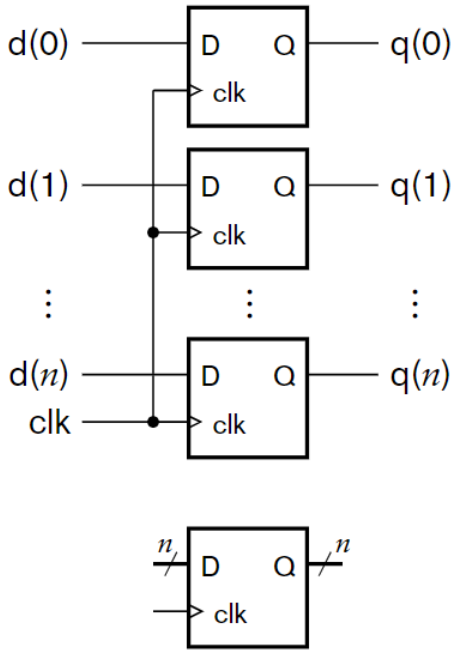


Kuva 3. D-kiikku [4].



Kuva 4. D-kiikun ajoituskaavio [4].

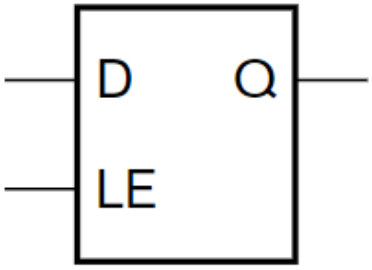
Useimmissa digitaalisissa piireissä D-kiikkuja ei käytetä yksittäisinä muisteina, vaan niistä muodostetaan ryhmiä, joihin voidaan säilöä useampia bittejä. Näitä ryhmiä kutsutaan rekistereiksi, ja niissä jokainen kiikku varastoi osan informaatiosta. Tällä tavalla toimiva rekisteri päivittyy jokaisella nousevalla kellonjaksolla, mikä tarkoittaa, että jokainen kiikku varastoi sen hetkisen tulon omaan muistiinsa. [4]



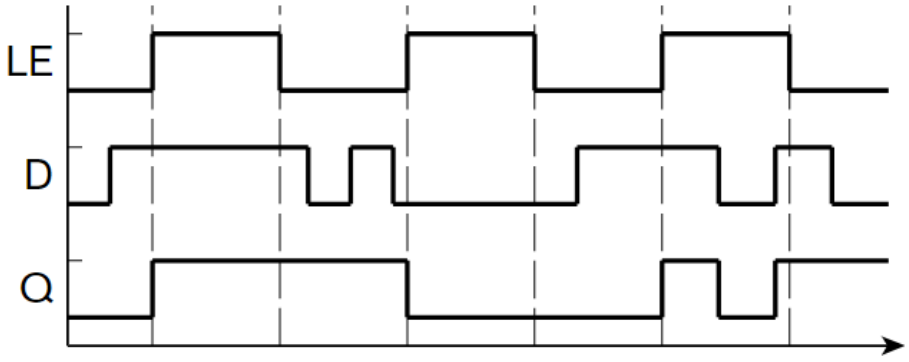
Kuva 5. D-kiikuista rakennettu rekisteri (yllä), ja rekisterin piirisymboli (alla) [4].

2.2.2 D-salpa

Kiikkujen lisäksi, joissakin järjestelmissä käytetään erilaisia sekvenssielementtejä, joita kutsutaan salvoiksi (engl. Latch). Salpojen ajoitus tiedon tallentamiseen toimii hieman eri tavalla poiketen kiikkujen ajoituksesta. D-salpa seuraa sen tulon D tilaa, silloin kun salvan ”enable” eli sallimistulo (engl. Latch Enable) on päällä, ja silloin salvan tila näkyy sen lähdössä Q, tällöin salpa toimii kuten kiikku. Mutta toisin kuin kiikulla, juuri ennen signaalin muuttumista, varastoitu arvo pysyy sekä salvassa että lähdössä, eli salvan lähtö ”salpautuu”, kun D-salvan aktivointisignaali menee nollaan. Tämän käyttäytymisen näkee hyvin kuvasta 7. Tällaista salpaa kutsutaan läpinäkyväksi salvaksi, koska sen läpi kulkee tieto, kun salvan aktivointisignaali on päällä. [4]



Kuva 6. Salpa [4].



Kuva 7. Salvan ajoituskaavio [4].

Vaikka salpapiirien suunnittelu ja toteutus on suhteellisen yksinkertaista, ne ovat nykyisin aika harvinaisia erilaisissa sekvenssipiireissä. Tämä johtuu salpakomponenttien vaatimista kellosignaaleista, jotka eivät ole päällekkäisiä. Toinen vaikeuttava asia on fakta, että niiden läpi kulkeva tieto voi virrata salpojen läpi läpinäkyvästi, mikä vaikeuttaa monimutkaisien järjestelmien suunnittelun oikeanlaisilla ajoituksilla digitaalisissa piireissä. Tärkeää on kuitenkin ymmärtää ja pohtia, kuinka salpakomponentit voivat syntyä vahingossa erilaisissa RTL-malleissa, sillä se on yleinen suunnitteluvirhe, ja sen kanssa on oltava erittäin huolellinen, kun RTL-malleja syntetisoidaan. [4]

3 SYNTETISOITAVAT MALLIT

Syntetisointia käytetään muuttamaan RTL-mallit sellaiseen porttitason abstraktioon, joka sisältää kaikki mallin portit, muistit ja muut komponentit yhdessä suunnitelmassa. Tässä osiossa selvitetään eri RTL-mallien vaatimuksia, jotta ne voidaan syntetisoida oikein. HDL-kieliä on monia, mutta eniten käytettyjä kieliä on kolme. Tässä työssä käytetään SystemVerilog kieltä RTL-mallien koodauksessa. Eri HDL-kielien erot näkyvät pääosin niiden syntaksissa sekä kirjoitustyylissä, ja lisää eroja kielien välillä on listattuna kuvan 8 taulukossa. [1]

Criterion	VHDL	Verilog	SystemVerilog
Synthesis support	yes	yes	growing
Parametrization & abstract modeling	good	poor	good
Type checking & scoping rules	strong	none	loose
Deterministic event queue mechanism	yes	not really	not really
Modeling of electric phenomena	9-valued	4-valued	4-valued
High-level verification support	limited	poor	excellent

Kuva 8. Eri HDL kielien teknisiä ominaisuuksia [1].

Kun tarkastellaan, mitä IC-piirin sisältä löytyy, voidaan sieltä löytää tietoa kuljettavia polkuja, erilaisia ohjaimia, tallennustilaa ja muita palasia, mitä ei näy ulospäin. Jokainen näistä mainituista lohkoista sisältää tuhansia eri logiikkaportteja, jotka puolestaan on luotu alkeellisimmista laitteista, kuten transistoreista. Tällaiset järjestelmät ovat jaoteltuja useisiin hierarkkisiin kerroksiin, sillä olisi epäkäytännöllistä määritellä ja dokumentoida piirit semmoisina, jossa on yhdessä tasossa miljoonia transistoreita. Sen sijaan suuremmat entiteetit koostuvat pienemmistä lohkoista, jotka ovat yhdistetty toisiinsa käyttäen väyläkomponentteja (engl. Bus) sekä johtimia (engl. Wire). Tällainen modulaarisuus sekä abstraktio parantaa piirien kuvausten laatua sekä ymmärtämistä. [1]

3.1 Kombinaatiologiikkamallien vaatimukset

Koska kombinaatiologiikkamallit ovat hyvin suoraviivaisia, ja niiden aritmeettiset sekä logiikkalaskutoimitukset ovat yksinkertaisia, ne voidaan yleensä ilmaista yhdessä käskyssä ilman haaroitusta. Tähän paras tapa on käyttää ”jatkuvaa asetusta” (engl. Continuous assignment). Tämä onkin SystemVerilog-kielen yksinkertaisimpia rakenteita muuttujan tai johtimen arvon määräämiseen. [1]

Assign-rakenne on oikeastaan vain yhdelle toteamukselle tehty prosessi. Tällöin vasemmanpuoleinen muuttuja päivittyy oikeanpuolen kohdemuuttujan vaihtaessa arvoa, ja seuraa sen arvoa jatkuvasti. On myös tilanteita, kun lauseeseen tarvitaan jonkinlainen ehto. Tätä

rakennetta voi myös käyttää ehdollisissa tilanteissa, jolloin koko ehtolause saadaan kirjoitettua samaan assign-rakenteeseen. Tällainen tilanne nähdään kuvan 9 rivillä 3. [1]

```
assign Today = Friday;                              (1)

assign Output = Input & A;                          (2)

assign Output = ADD ? (Input + A) : (Input - A);    (3)
```

Kuva 9. Assign-rakenteen käyttö

Kombinaatiologiikkaa voidaan mallintaa RTL-koodissa myös käyttäen always-ilmaisua. Tämä tapahtuu prosessilohkon sisällä, ja se mahdollistaa useamman rivin käytön toisin kuin assign-komento. Samanaikaisia prosesseja voidaan ilmaista monella tapaa, ja näistä kombinaatiologiikkaa edustaa always_comb sekä always lohkot.

Näitä lohkoja käytettäessä herkkyyslista ei saa sisältää kellon reunatapahtumia kuten positiivista reunaa tai negatiivista reunaa. Kaikissa kombinaatiologiikkaa sisältävissä lauseissa täytyy myös muistaa käyttää estomääritystä (=) (engl. Blocking assignment), ja samaan aikaan olla käyttämättä estotonta määritystä (<=) (engl. Nonblocking assignment), jota käytetään sekvenssilogiikkamallien suunnittelussa. [5]

```
always @ (in1 or in2)
    out = in1 + in2;

always @ * begin
    tmp1 = a & b;
    tmp = c & d;
    z = tmp1 | tmp2;
end

always_comb begin
    out = in1 + in2;
    counter = counter + (in1 - in2);
end
```

Kuva 10. Always-lohkojen käyttö kombinaatiologiikkamalleissa

Always-rakennetta voidaan hyödyntää lukuisissa kombinaatiologiikkamallien suunnitteluissa. Kuvan 10 ensimmäinen always-lohko sisältää herkkyyslistan, jonka perusteella lohko päivittyy. Herkkyyslistan muuttujat ja niiden toteutuminen määräävät, milloin lohkon sisältö toteutetaan. Keskimmäinen always-lohko on esimerkki implisiittisestä herkkyyslistasta. Implisiittinen herkkyyslista lisää kaikki lohkon sisällä olevat muuttujat herkkyyslistan jäseniksi. Always_comb-lohkoa voi ajatella säiliönä pienelle ohjelmalle, joka generoi sisältämälleen piirille totuustaulun. [1][5]

3.2 Sekvenssilogiikkamallien vaatimukset

Toisinkuin kombinaatiologiikka, sekvenssilogiikkaa mallinnetaan aina always-prosessilla, jossa on yksi tai useampi muuttuja herkkyyslistassa. Näiden mallien herkkyyslistassa käytetään HDL-kielen varattuja sanoa posedge tai negedge, jotka kertovat mallille, tapahtuuko sekvenssilogiikka kellon nousevalla vai laskevalla reunalla. Näitä lohkoja käytetään mallintamaan erityisesti kiikkuja, sekä muita reunaliipaistuja digitaalisia piirejä, kuten tilakoneita. [1][5]

Sekvenssilogiikkamalleissa on tärkeä käyttää estomäärityksen (=) sijaan estotonta määritystä (<=). Tällä estottomalla määrityksellä viitataan tiedon tallentumiseen jonkinlaiseen tallennuskomponenttiin, kuten rekisteriin. [5]

```
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        out <= '0;
    else
        out <= in1;
end
```

Kuva 11. Always_ff lohkon käyttö.

Kuvassa 11 nähdään hyvin yksinkertainen esimerkki sekvenssilogiikan mallinnuksesta käyttäen always_ff lohkoa. Tässä esimerkissä käytetään reunaliipaistua rekisteriä ”out”, sekä asynkronista nollaustuloa ”rst_n”, jolla rekisterin tila voidaan pakottaa nollaksi riippumatta sen kellosta. Herkkyyslistan sisältämä posedge clk reagoi, jokaisella kellon nousevalla reunalla, ja negedge rst_n reagoi jokaisella nollaustulon laskevalla reunalla. Tässä mallissa oletetaan, että nollaustulo on aktiivinen alhaalla (engl. Active low), joka tarkoittaa, sitä että reset-signaali on päällä, silloin kun sen arvo on nolla. [5]

Kuten kombinaatiologiikkamallien tapauksessa, sekvenssilogiikan always-prosessien sisällä voidaan käyttää useita rivejä ja ehtolauseita, ja näiden pohjalta luoda monimutkaisien piirien malleja. Sekvenssilogiikkaa käytetään esimerkiksi tilakoneiden mallinnuksessa, sillä eri tilat voidaan päivittää rekistereihin eri kellon hetkillä. Esimerkki yksinkertaisesta tilakoneen mallista löytyy kuvasta 12.

```
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= '0;
    end
    else begin
        if (sel1)
            state <= A;
        else if (sel2)
            state <= B;
        else
            state <= IDLE;
    end
end
```

Kuva 12. Tilakoneen mallintaminen always-lohkoa käyttäen.

3.3 Virheelliset mallit

Jotta RTL-mallit voitaisiin syntetisoida, ne eivät saa sisältää virheellisiä rakenteita tai lohkoja. Rekisterisiirtotason koodaus ei ole aina välttämättä täysin suoraviivaista, ja joidenkin mallien ymmärtäminen voi olla haastavaa. Seuraavat esimerkit osoittavat, millaisia virheellisiä malleja RTL-koodista voi saada aikaan ilman, että itse syntaksissa olisi mitään virheellistä.

3.3.1 Kombinaatio- ja sekvenssilogiikan sekoitus

Yksi tapa tehdä virheellinen RTL-malli on käyttää samassa lohkossa sekä kombinaatiologiikkaa sekä sekvenssilogiikkaa. Tämä voidaan tehdä monella tapaa, ja niistä kaksi esimerkkiä löytyy kuvasta 13. [6]

Koska sekvenssilogiikkamallit ovat määriteltyjä yksinkertaisella always-lohkolla, jossa käytetään kellon reunaa kuten posedge muuttujaa herkkyyslistassa, sitä ei saa sekoittaa kombinaatiologiikan kanssa, kuten on tehty kuvan 13 ensimmäisessä mallissa. Toinen virheellinen tapa sekoittaa kombinaatiologiikkaa sekä sekvenssilogiikkaa on käyttää sekaisin kombinaatiologiikassa käytettävää estomääritystä (=) sekä sekvenssilogiikassa käytettävää estotonta määritystä (<=), kuten on tehty kuvan 13 alemmassa mallissa. Tärkeää on siis muistaa pitää sekvenssi- ja kombinaatiologiikka erillään, ja muistaa niiden eroavaisuudet. [6]

```
always @ (posedge or in2)                (1)
    out = in1 + in2;

always @ (in1 or in2 or sel) begin       (2)
    out = in1;
    if (sel)
        out <= in2;
end
```

Kuva 13. Virheellinen logiikkamallien sekoitus

3.3.2 Logiikkaa käytetty nollaustulolle tai kellolle

Digitaalisen piirin kellosignaali sekä nollaustulo eli reset-signaali eivät ole mitä tahansa signaaleja, ja niiden käyttöön liittyy sääntöjä, joita täytyy noudattaa huolella. Tämä tarkoittaa pääasiassa sitä, että kellosignaaleja kuuluu käyttää vain kellon rekisterien kanssa, ja sama koskee nollaustulosignaaleja. Kaikki häiriöt kello- sekä reset-signaaleissa ovat kohtalokkaita, ja ne aiheuttavat piiriin erilaisia ajoitusongelmia. [6]

Kuva 14 esittää kaksi esimerkkiä virheellisistä logiikkamalleista, jotka käyttävät nollaustulosignaalia (rst_n) sekä kellosignaalia (clk) yhdessä logiikan kanssa. Kellosignaalin käyttäminen logiikassa aiheuttaa häiriötä, joka saa aikaan ei-toivottua sekä odottamatonta datan näytteistystä. Reset-signaalin käyttäminen logiikassa sen sijaan saa aikaan odottamattomia ja vahingossa tapahtuvia nollauksia muisteissa. [6]

```
assign out = a && rst_n;

always_comb begin
    if (b || clk)
        out = a;
end
```

Kuva 14. Kello- ja reset-signaalin käyttäminen logiikassa.

3.3.3 Kelloalueiden sekoitus

Joissakin digitaalisissa piireissä käytetään useampaa kuin yhtä kelloa. Tämä tarkoittaa sitä, että kelloilla voi olla eri taajuudet sekä lähteet, joista kellot tulevat. Tämän perusteella kellot ovat siis asynkronisia, ja niitä ei saa sekoittaa toisiinsa suoralla polulla. Tätä kutsutaan digitaalitekniikassa nimellä ”Clock Domain Crossing” eli CDC, joka tarkoittaa signaalien kulkemista eri kelloalueiden välillä. [6]

Kuva 15 on esimerkki kahden eri kelloalueen kellosignaalien sekoituksesta. Siinä virheellinen malli syntyy, kun kellon ”clk1” mukaan päivittyvä rekisteri ”a” määrittää kombinaatiologiikalla kellon ”clk2” mukaan päivittyvää rekisteriä ”next_b”.

```
always_ff @(posedge clk1)
    a <= next_a;

always_ff @(posedge clk2)
    b <= next_b;

always_comb begin
    next_b = a;
end
```

Kuva 15. Eri kelloalueiden kellojen sekoitus.

3.3.4 Salpojen päättely

Salpojen eli latchien rakentuminen on yleinen ongelma syntetisoinnissa, ja niiden välttäminen on tärkeää, jos haluaa suunnitella RTL-malleja oikein. Salvat syntyvät, kun RTL-malli haluaa laittaa tietyn arvon muistiin, vaikka näin ei välttämättä haluttaisi tai se ei ollut koodin tarkoitus. Tämä voi tapahtua todella monella tavalla ja niiden välttäminen on tärkeää syntetisoinnin kannalta. [6]

Ensimmäinen syy salpojen päättelyyn syntetisointivaiheessa on kombinaatiologiikan ehtolauseen vajaavaisuus. Yleinen salvan päättely tapahtuu, jos käytetään if-lausetta ilman, että koodi sisältää tälle ehdolle else-lausetta. Täytyy siis varmistaa, että jokaiselle if-lauseelle löytyy oma else-lauseensa koodin lopusta. [6]

RTL-malleissa käytetään usein case-rakennetta, jolla voidaan vertailla valitun muuttujan tilaa eri vaihtoehtoihin. Periaate on sama kuin muilla ehtolauseilla, mutta rakenne erilainen, ja sen käyttö on joissakin tilanteissa suotavampaa kuin esimerkiksi if-lauseet. Mutta, jos case-rakenteen vaihtoehdoissa ei ole kaikkia saatavilla olevia vaihtoehtoja, luokitellaan se virheeksi ja siihen päätellään salpa. Samaan aikaan case-rakenteessa voi syntyä virhe, jos sille ei ole määritelty oletustilaa (engl. Default state) eli sellaista vaihtoehtoa, joka toteutuu, kun mikään muu vaihtoehto ei toteudu. Nämä molemmat virheet on esitelty kuvassa 16. [6]

```
logic [1:0] state;

always_comb begin
    case (state)
        2'b00: begin
            next_state = 2'b01;
        end
        2'b01: begin
            next_state = 2'b00;
        end
    endcase
end
```

Kuva 16. Virheellinen case-rakenne.

Viimeinen virheellinen tapa luoda salpakomponentteja on jättää oletusarvojen määrittäminen pois. Kuvan 17 tapauksessa muuttujat a, b ja c saavat arvot 1 tai 0 riippuen state-muuttujasta, mutta ne eivät saa mitään arvoa, silloin kun niiden ehto ei toteudu. Tämä virheellinen rakenne täytyy ottaa huomioon myös kaikilla muilla ehdollisilla rakenteilla, esimerkiksi kun käytetään if-else-rakennetta. [6]

```
always_comb begin
/*
    Oletusarvot täytyy määritellä tässä
    a = 0;
    ...
    ...
*/
    case (state)
        state1: begin
            a = 1;
        end
        state2: begin
            b = 0;
        end
        state3: begin
            c = 1;
        end
    endcase
end
```

Kuva 17. Oletusarvojen puuttuminen kombinaatiorakenteesta.

4 ESIMERKKIMALLIEN TARKASTAMINEN

Yksi tärkeimpiä ja haastavimpia työvaiheita digitaalisten järjestelmien kehityksessä on RTL-verifikaatio. Tässä vaiheessa saadaan vahvistus siihen, että digitaalinen piiri on toiminnallisesti oikein tehty, ja siitä löytyvät virheet huomataan sekä korjataan. RTL-verifikaatiossa käytetään erityyppisiä työkaluja varmistamaan suunnitellut mallit. [7]

Esimerkkimallien tarkastamiseen käytetään Cadence JasperGold SuperLint työkalua. Lint-työkalut ovat yksiä tehokkaimpia työkaluja designien tarkistamiseen, sillä ne tarkistavat sekä staattisen että formaalin analyysin tehokkaimmilla tavoilla. Lint-työkaluja käytetään tarkistamaan mahdolliset eroavaisuudet synteesin sekä simulaation välillä. Näillä tarkistuksilla siis voidaan varmistaa, että RTL-mallit syntetisoituvat oikein, eikä virheitä pääse syntymään. Staattiset analyysit näkevät RTL-koodin sekä syntetisoidun esityksen koodista, ja ne pystyvät näiden perusteella tunnistamaan eroavaisuuksia. Tutkimalla syntetisoitua mallia, työkalu pystyy tunnistamaan, missä kohdassa RTL-koodia tapahtuu virhe, tai millä tavalla logiikka on kirjoitettu väärin. [7][8]

RTL-lint-työkalut tutkivat HDL-koodia eri näkökulmista, kuten syntetisoitavuuden, simulaation, testattavuuden sekä käytettävyyden puolesta. Eri työkaluilla on erilaisia sääntöjä, minkä mukaan ne pystyvät tunnistamaan virheelliset koodit. Näillä ohjeilla voidaan myös tarkistaa nimeämiskäytäntöjä, ja koodin luettavuutta. [8]

4.1 Virheellisten mallien tarkistus työkalulla

Esimerkkikoodina käytetään Digitaalitekniikka 2 kurssilla esiteltyä tilakoneen ja datapolun vuorovaikutukseen keskittyvää RTL-koodia. Esimerkkinä käytettävä RTL-koodi ja sen sisältämät mallit syntetisoituvat oikein ja lint-työkalu ei löydä syntetisoitumiseen vaikuttavia virheitä. Ideana on kuitenkin muokata RTL-malleja niin, että lint-työkalu löytää niistä virheitä, ja tutkia työkalun ilmoittamia virheilmoituksia.

4.1.1 Kombinaatio- ja sekvenssilogiikan sekoitus

Ensimmäisenä esimerkkinä tarkastellaan tilannetta, missä datapolun esimerkkikoodiin muokataan always-lohko, joka käyttää samassa prosessissa sekä sekvenssilogiikkaa, että kombinaatiologiikkaa. Tässä esimerkissä käytetään virheellisesti estomääritystä (=) rekisteripankin määritykseen kuvan 18 mukaisesti.

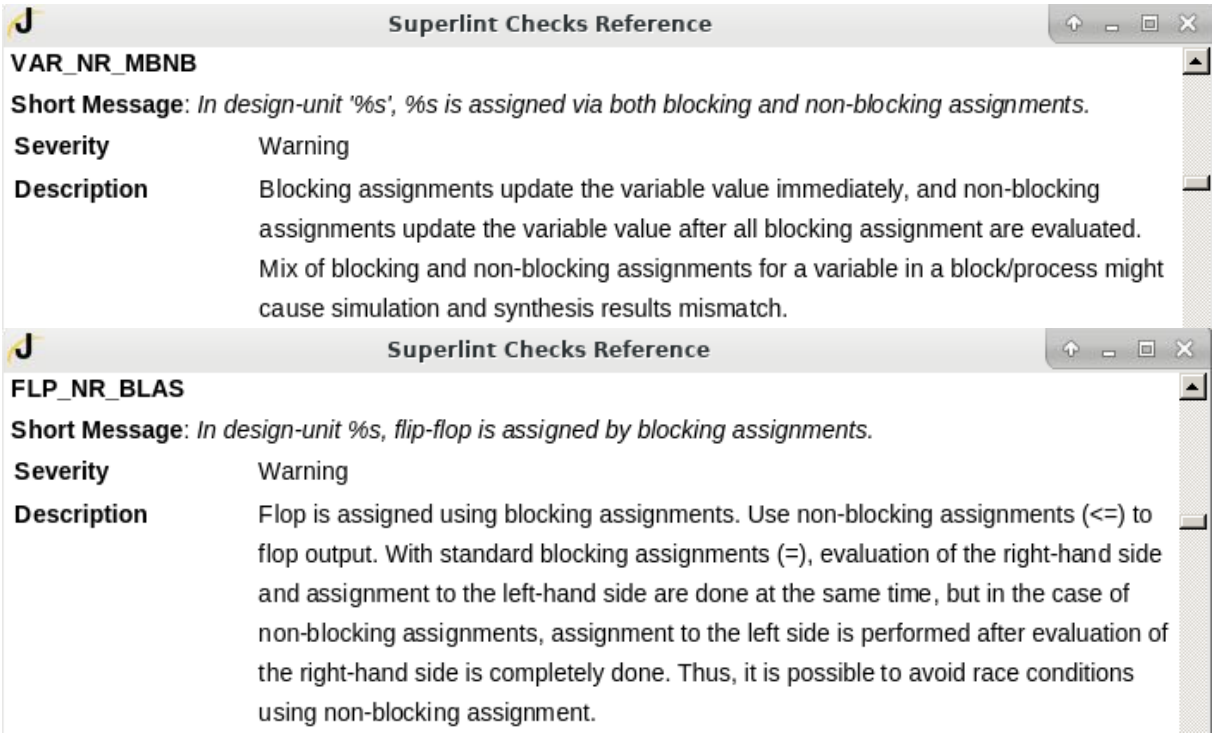
```
always_ff @(posedge clk or negedge rst_n)
  begin : rbank
 if (!rst_n)
   rb_r <= '0;
 else
   if (we_in)
     rb_r[rsel_in] = busd;
 end : rbank
```

Kuva 18. Virheellinen malli 1

Tarkistaessa tätä RTL-mallia, voidaan huomata kuinka SuperLint-työkalu huomaa virheen ja ilmoittaa siitä kahdella eri tavalla, joka nähdään kuvassa 19. Virheilmoituksista nähdään, että työkalu ilmoittaa ensimmäisenä siitä, että always-lohkossa käytetään sekä estomääritystä (=), että estotonta määritystä (<=), ja toisena siitä, että kiikku on määritetty estollisella määrityksellä, mikä on myös virhe. Työkalu osoittaa myös suoraan sille koodiriville, missä virheet tapahtuvat. Kuvasta 20 löytyy myös työkalun antamat virheilmoituksien kuvaukset. Näistä löytyy kattavasti tietoa, miksi RTL-malli on koodattu väärin. Työkalu antaa myös esimerkkejä virheellisestä koodista, jota voi vertailla omiin malleihin.

```
Category: SIM_SYNTH (1)
   Tag: VAR_NR_MBNB (1)
      "In design-unit 'dp', rb_r[rsel_in] is assigned via both blocking and non-blocking assignments"   example/dp.sv
Category: SYNTHESIS (4)
   Tag: FLP_NR_BLAS (1)
      "In design-unit dp, flip-flop is assigned by blocking assignments"                               example/dp.sv
```

Kuva 19. Lint-työkalun virheilmoitukset mallista 1.



Kuva 20. Virheilmoituksien kuvaukset mallille 1.

4.1.2 Salpojen päättely

Kuten aikaisemmin mainittu, salvat saavat aikaan ajoitus ongelmia, kun ne generoituvat vahingossa. Tätä salpojen päättelyä halutaan siis välttää RTL-mallien koodaamisessa, ja se onnistuu käyttämällä lint-työkalua.

Ensimmäisessä latch-esimerkissä käytetään koodia, jonka ehtolause on vajavainen. Tässä tapauksessa if-lauseelle ei ole annettu else-lausetta jatkeeksi, vaan se on kommentoitu ulos, jolloin työkalut ja simulaatiot eivät ota sitä huomioon osana RTL-koodia (kuva 21). Tämän takia syntetisoinnissa tapahtuu virheitä.

```
// Zero flag

if (alu == 0)
  z_out = '1;

/*else
  z_out = '0;*/

// Negative flag

if (alu < 0)
  n_out = '1;
else
  n_out = '0;
```

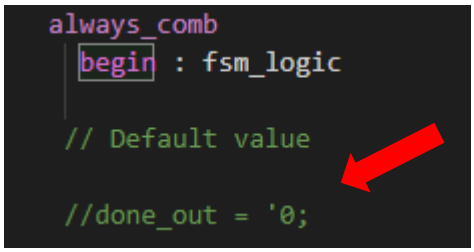
Kuva 21. Virheellinen malli 2

Suorittaessa SuperLint-työkalua, huomataan että virheellinen RTL-malli saa aikaa kaksi virheilmoitusta, jotka näkyvät kuvassa 22. Kuten aikaisemmin on todettu, vajavainen ehtolause saa aikaan haluamattomia salpoja, jotka pilaavat designin.

```
Category: SYNTHESIS (5)
  Tag: ALW_NO_COMB (1)
    "The variable 'z_out' models a Latch in an 'always_comb' block"        example/dp.sv
  Tag: LAT_NR_BLAS (1)
    "In design-unit dp, latch is assigned by blocking assignments"         example/dp.sv
```

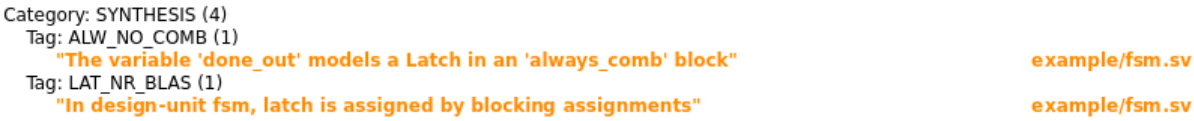
Kuva 22. Lint-työkalun virheilmoitukset mallista 2.

Kuten aikaisemmin todettiin, latch-virheitä voi tulla muillakin tavoin kuin vääränlaisilla ehtolauseilla. Yksi virheellinen tapa on jättää pois oletusarvot muuttujille esimerkiksi always-lohkon sisällä, ja tästä esimerkki löytyy kuvasta 23. Tässä virheellisen mallin esimerkissä always-lohkon ”done_out” oletusarvon 0 määrittäminen on kommentoitu pois koodista.

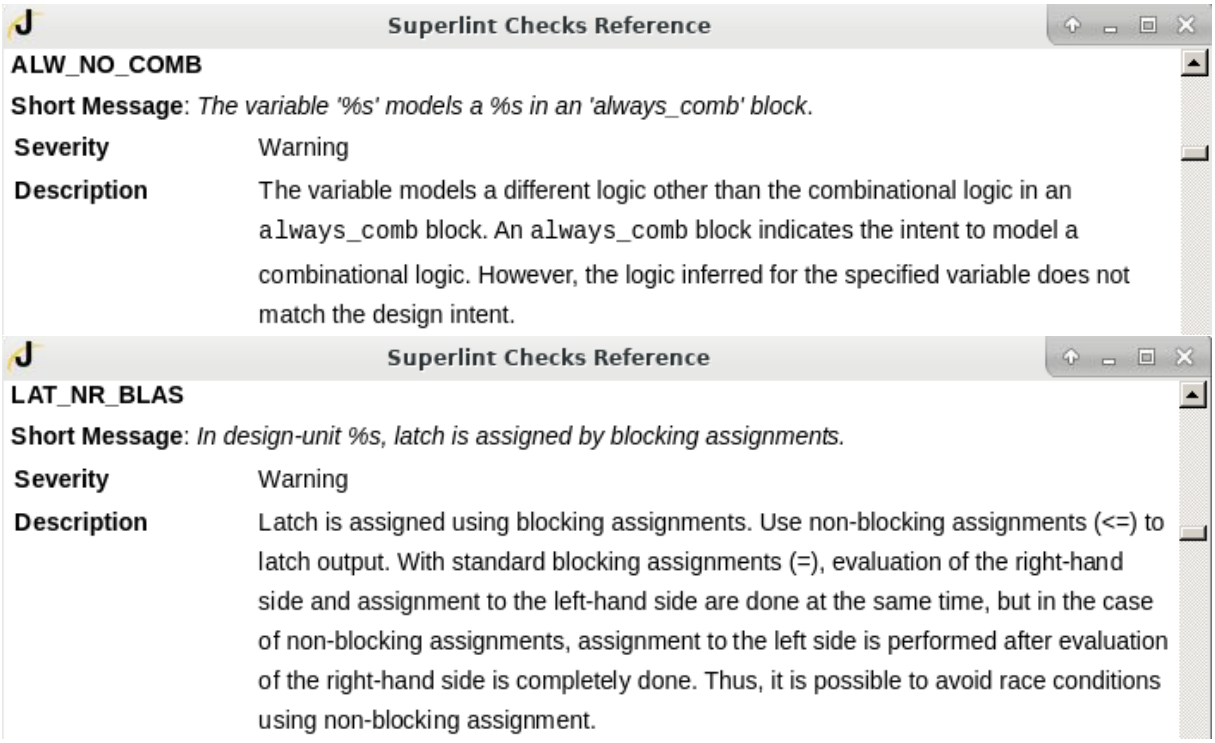


Kuva 23. Virheellinen malli 3

Virheellisen mallin 3 virheilmoitukset ilmoitetaan samoilla ilmoituksilla kuin virheellisen mallin 2 virheilmoitukset. Näiden virheilmoitusten eli ”ALW_NO_COMB” sekä ”LAT_NR_BLAS” kuvaukset löytyvät alempaa kuvasta 25.



Kuva 24. Lint-työkalun virheilmoitukset mallista 3.



Kuva 25. Virheilmoituksien kuvaukset malleille 2 ja 3.

5 POHDINTA

Tämän kandidaatin työn tavoitteena oli RTL-mallien tutkiminen, sekä niiden vaatimusten tarkastelu, niin että RTL-koodin saa syntetisoitavaan muotoon. Työssä käytiin läpi yksityiskohtaisesti erilaisia RTL-malleja, ja niiden pohjalta tehtyjä virheellisiä versioita esimerkkikoodia hyödyntäen.

Teoriaosuuksissa on hyvin yksinkertaista, mutta aiheeseen täydellisesti liittyvää asiaa, jota ei olisi voinut jättää kirjoittamatta. Kombinaatio- sekä sekvenssilogiikan perusteet on hyvä käydä läpi, jotta RTL-koodilla kirjoitetut kombinaatio- ja sekvenssilogiikkamallit ovat helpommin ymmärrettävissä, eikä niissä tule väärinymmärryksiä. Tämä myös auttaa ymmärtämään virheellisten mallien ongelmia.

Tietenkin työn monipuolisuutta voisi vieläkin lisätä esimerkiksi käymällä monimutkaisempia RTL-malleja läpi, mutta en kokenut sitä välttämättä kovin tarpeelliseksi. Myös erilaisten mallien virheitä löytyy paljon enemmän, ja käyttämällä lint-työkalua niitä voitaisiin tutkia lisää.

Koen että tekemäni kandidaatintyö auttaa minua itseänikin hahmottamaan paremmin rekisterisiirtotason mallien vaatimuksia, ja tästä edespäin käytän vaatimusten noudattamiseen varmasti enemmän aikaa omaa RTL-koodia kirjoittaessani.

Mielestäni työ onnistui tavoitteiden mukaisesti, ja se kokosi hyvin rekisterisiirtotasoon liittyvien mallien erilaisia vaatimuksia. Teoriaosuuksien havainnollistaminen käytännön osuudella Cadence JasperGold SuperLint-työkalua käyttäen oli mielestäni hyvä lisäys kandidaatin työhön. Tämän avulla työstä saatiin käytännönläheisempi, sillä virheellisiä RTL-malleja tutkimalla päästään lopputulokseen, että syntetisoitavien RTL-mallien koodaussääntöjä täytyy todellakin noudattaa.

6 YHTEENVETO

Syntetisoitavien RTL-mallien koodaussäännöt viittaavat niihin vaatimuksiin, joilla rekisterisiirtotason malleista saadaan tehtyä porttitason abstraktio, jota voidaan käyttää digitaalisten piirien suunnittelun myöhemmissä työvaiheissa. Syntetisoinnin avulla kirjoitetusta koodista saadaan tehtyä suunnitelma, joka lopulta syntetisoituu logiikkaporteiksi sekä rekistereiksi.

Kandidaatin työssä perehdyttiin HDL-synteesin merkitykseen ja siihen, miksi tätä tehdään. Lisäksi kerrotaan, kuinka mallinnus on helpottanut matalan tason abstraktioiden suunnittelua, ja kuinka tärkeää se on nykypäivänä. Digitaalisten piirien perusteet on myös käyty läpi huolellisesti teoriatasolla.

Työssä käytiin läpi niin kombinaatiologiikan perusteet, kuin sekvenssilogiikankin perusteet. Nämä teoriaosuudet vaikuttavat hyvin yksinkertaisilta, mutta niiden ymmärtäminen on todella tärkeää. Kombinaatiologiikka osuus sisältää esimerkkejä Boolen lauseista, operaatioista sekä totuustauluista. Lisäksi käydään läpi perus logiikkaporttien merkitykset ja toiminta. Sekvenssilogiikka osuus kertoo kuinka sekvenssilogiikkakomponentit toimivat, ja mitä ominaisuuksia niillä on. Tämä osuus sisältää myös ajoituksellisia kaavioita, jotka hahmottavat hyvin sekvenssilogiikan toimintaa.

Myöhemmin käydään läpi, mitä RTL-mallien vaatimukset käytännössä tarkoittavat, ja niistä esitetään lukuisia esimerkkejä. Samaan yhteyteen on lisätty virheelliset RTL-mallit, jotka eivät virheidensä takia voi syntetisoitua. Ja näitä malleja voidaan hyvin verrata vaatimusten malleihin, joiden on tarkoitus syntetisoitua oikealla tavalla.

Käytännön osuus sisältää Cadence JasperGold SuperLint-työkalun käyttöä. Tällä voidaan tarkastaa virheitä erilaisista RTL-malleista todella kätevästi. Yksi tärkeimpiä työkalun käyttökohteita onkin tarkastaa, syntetisoituuko kirjoitettu koodi oikein. Käytännön osuus perustuu esimerkkikoodiin, jota on muokattu tarkoituksella niin, että koodi ei syntetisoidu oikein, jolloin voidaan tarkastella työkalun antamia virheilmoituksia. Kaikki virheelliset mallit perustuvat aikaisemmin työssä esiteltyihin RTL-mallien vaatimuksiin.

7 LÄHDELUETTELO

[1] Kaeslin H. A. (2015) Top-down digital VLSI design: From architectures to gate-level circuits and FPGAS, s. 179-300

[2] Cofer R.C. & Harding B.F. (2006) Rapid System Prototyping with FPGAs : Accelerating the Design Process, Amsterdam, s. 103-125

[3] Semiconductor Engineering, (luettu 09.1.2023), RTL (Register Transfer Level), An abstraction for defining the digital portions of a design. URL: https://semiengineering.com/knowledge_centers/eda-design/definitions/register-transfer-level/

[4] Ashenden P. J. (2008) Digital Design (Verilog) : An Embedded Systems Approach Using Verilog, Amsterdam

[5] "IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis," in IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1 , vol., no., pp.1-116, 18 Dec. 2005, doi: 10.1109/IEEESTD.2005.339572.

[6] Teman A. (2021) How to code Synthesizeable RTL. URL: https://www.eng.biu.ac.il/temanad/files/2021/12/Synthesizeable-RTL-2021-22.pdf

[7] Yadav A. & Jindal P. & Basappa D. (2020) Study and Analysis of RTL Verification Tool, IEEE Students Conference on Engineering & Systems (SCES), Prayagraj, India

[8] Yeung P. & Choi S. (2009) Advanced static verification for SoC designs, International SoC Design Conference, Busan, South Korea