Tampereen ammattikorkeakoulu

UVM-testipenkin koodikattavuuden nostaminen

Kimmo Saranpää

OPINNÄYTETYÖ
Kesäkuu 2022

Tieto- ja viestintätekniikka
Sulautetut järjestelmät

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikan tutkinto-ohjelma
Sulautetut järjestelmät

SARANPÄÄ, KIMMO:
UVM-testipenkin koodikattavuuden nostaminen

Opinnäytetyö 23 sivua
Kesäkuu 2022

Opinnäytetyön tavoitteena oli viedä yrityksen keskeneräinen IP-lohkon (Intellectual Property) testipenkki loppuun. Testipenkki on toteutettu SystemVerilog-kielellä, UVM (Universal Verification Methodology) -metodologiaa noudattaen. Opinnäytetyöraportissa käsitellään verifiointia ja koodikattavuutta sekä näihin tarkoitettuja työkaluja ja toimitapoja. Tarkoituksena oli tutustua UVM:ään, verifiointimenetelmiin sekä niiden työkaluihin ja samalla tehdä valmiiksi vajavainen testipenkki. Huomion kohteena oli erityisesti koodikattavuus, joka on eräänlainen mittari sille, milloin kohde on verifioitu. Raportissa käsitellään myös UVM:ää, joten lukijan pitäisi saada perustiedot verifioimisesta, koodikattavuudesta ja UVM:stä opinnäytetyön lukemisen jälkeen.

Testipenkin valmiiksi tekeminen tarkoittaa tässä sitä, että lohko, johon testipenkki on tehty, verifioidaan loppuun. Projektissa oli tarkoitus saada koodikattavuus tasolle, jolla voitaisiin sanoa, että lohko on verifioitu. Koodikattavuuden kasvattaminen oli suurimmalta osin testien parantelemista ja sekvenssien lisäämistä. Lohkoa ei loppujen lopuksi saatu verifioitua aivan loppuun asti. Kaikkia testejä ei ehditty viimeistellä täydellisesti, mutta koodikattavuutta saatiin silti nostettua niin, että kattavuus on opinnäytetyön kirjoitushetkellä 81,84 %. Kattavuudesta lohko voidaan käyttötarkoituksen perusteella todeta riittävän verifioiduksi.

Koodikattavuus on tärkeä työkalu verifioinnin etenemisen ja valmiuden mittaamiseksi, mutta sen ei kaikissa sovelluksissa tarvitse olla 100 %, vaikka siihen pyritäänkin. Täydellinen koodikattavuus vaatii aikaa, resursseja ja tulee sitä vaikeammaksi, mitä lähemmäs 100 %:a päästään, joten tietyssä vaiheessa on halvempaa ja helpompaa todeta lohko verifioiduksi aikaisemmin.

Asiasanat: UVM, koodikattavuus, verifiointi

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering
Embedded Systems

SARANPÄÄ, KIMMO:
Raising Code Coverage in a UVM Testbench

Bachelor's thesis 23 pages
June 2022

The objective of this thesis was to finish an already existing testbench for a company's IP block, a reusable unit of logic that is the intellectual property of one party. The testbench is made using SystemVerilog and UVM (Universal Verification Methodology) methodology. This thesis also goes over verification and code coverage along with some of the tools and procedures for them. The motivation for this project was to learn UVM and verification, their tools and to finish a uncomplete testbench. The focus of this thesis is code coverage which can be used to measure if a target is verified.

The testbench was continued by finding which parts of it were unfinished or missing. The main changes included adding sequences, improving tests, and adding code coverage components. At the time of writing this thesis, the IP block is not fully verified. The coverage only reached 81.84 %. At this point the block could be considered verified since most blocks do not reach 100% coverage. However, the block can be verified even further with relatively little effort.

Keywords: UVM, code coverage, verification

SISÄLLYS

ERITYISSANASTO

SoC	System-on-chip, järjestelmäpiiri, kaikki yhden järjestelmän tarvitsema toimivuus yhdessä piirissä
ASIC	Application-spesific integration circuit, tiettyyn tarkoitukseen tarkoitettu mikropiiri
FPGA	Field programmable gate array, uudelleenohjelmoitava logiikkapiiri
UVM	Universal Verification Methodology, standardoitu metodologia verifioimiseen
DUT	Device under test, testattava laite
EDA	Electronic design automation, elektroniikan suunnitteluautomaatio, työkaluja, jotka tekevät piirien suunnittelusta huomattavasti helpompaa
IP-lohko	Intellectual property, kolmannen osapuolen omistama logiikkapiiri
RAL	Register abstraction layer, rekistereihin pääsyä helpottavia pohjaluokkia UVM:n sisällä

1 JOHDANTO

System-on-chip tai SoC lyhyesti on piirilevy, jossa on yhdistettynä mm. suoritin, muisti, grafiikkaprosessori ja muita tarvittavia lisäosia tarpeen mukaan. SoC:illa voidaan tarkoittaa useampaa piirilevyä, joilla on omat tarkoituksensa ja valmistus- ja suunnittelumenetelmät. Esimerkiksi ASIC:it (Application-Specific Integration Circuit) voivat olla SoC:eja riippuen siitä onko piirilevyssä prosessoria. Nimestä voi päätellä että ASIC:it ovat usein suunniteltu erityisesti yhteen tehtävään ja niin ne voidaan tehdä tehokkaiksi ja pienikokoisiksi. Toisin kuin FPGA:t (Field Programmable Gate Arrays), ASIC:eja ei voi enää uudelleenohjelmoida piirin valmistamisen jälkeen, joten näiden piirien suunnittelussa on tärkeää varmistaa se, että suunniteltu piiri sisältäisi mahdollisimman vähän bugeja, ja etteivät läpimenevät bugit ole vakavasti toimintaa häiritseviä. ASIC:eja on kallista suunnitella, mutta koska ne ovat juuri tiettyyn tehtävään suunniteltuja, niitä on halvempi valmistaa suurissa määrissä

Koska ASIC:in kehitystyö on kallista ja aikaa vievää on tärkeää tietää, että ne toimivat halutulla tavalla ja ne ovat mahdollisimman virheettömiä ennen kuin ne menevät tuotantoon. Jos jo tuotannossa oleva piiri joudutaan suunnittelemaan bugin takia uudestaan, se on erittäin kallista yhtiölle. Tätä varten verifiointi on tärkeää, jotta päästäisiin siihen tavoitteeseen, että kun piiri menee valmistukseen niin sitä ei tarvitse analysoida, suunnitella ja rakentaa uudestaan.

2 VERIFIOINTI

Tässä luvussa tutustutaan verifiointiin. Mihin verifiointia tarvitaan ja mitä tekniikoita siihen käytetään.

2.1 Tavoite

Jos lohkossa tai piirissä on virheitä, ja se menee eteenpäin, niin virheen löytäminen ja korjaaminen myöhemmin voi olla hyvin paljon vaikeampaa ja siihen menee paljon enemmän aikaa. Pahimmassa tapauksessa virhettä ei huomata, ennen kuin piiri menee valmistukseen. Tässä vaiheessa virheen korjaaminen tulee paljon kalliimmaksi ja vaikeammaksi kuin jos se olisi huomattu ajoissa.

Verifioinnin tavoitteena on varmistaa, että kaikki virheet ja bugit saadaan huomattua ja korjattua ennen kuin lohkon tai piirin valmistaminen edistyy. Hyvän verifioinnin avulla saadaan varmistettua lohkojen ja piirien toimivuus ajoissa ja näin virheiden korjaamisesta tulee helpompaa ja halvempaa.

2.2 Ohjattu ja satunnainen verifiointi

Verifioinnissa testattavalle piirille ajetaan sisääntuloja, siitä tarkistetaan ulostulot, ja näitä ulostuloja verrataan referenssimalliin, jotta nähdään täsmäävätkö ulostulot. Tätä eri sisääntuloja ja parametrejä ohjaamalla voidaan optimoida testien tehokkuutta. Kaikista yleisimmin nykyään käytetään syötteen satunnaistamista, mikä tarkoittaa sitä, että kaikki DUT:ille menevä syöte ja sen rekisterit satunnaistetaan testiä varten. Kaikille satunnaisille arvoille pitää antaa rajat, joiden sisällä satunnaisten arvojen pitää olla, jotta vältetään turhat virheet ja verifiointi nopeutuu. (Lemiengre 2017)

Ohjatussa testaamisessa syötteen ja rekisterien arvot ovat verifioijan itsensä valitsemia. Tarkoituksena on se, että verifioija testaa ohjatusti SoC:in tai IP-lohkon toiminnan. Kun ohjelmoija itse päättää parametrit joita testauksessa käytetään niin testauksessa voi helpommin jäädä huomaamatta tiettyjä kulmatapauksia joissa olisi virhe. Toisaalta ohjelmoija itse voi tietää tärkeät tapaukset, jotka on testattava, joita satunnainen testaus ei välttämättä käy läpi. (Lemiengre 2017)

2.3 Koodikattavuus

Koodikattavuus on mitta siitä, kuinka monta prosenttia koodista on verifioitu toimivaksi. Sitä voidaan mitata usealla simulaattorilla. Koodikattavuutta ajaessa simulaattori ajaa kaikki testit ja katsoo kuinka monta prosenttia testattavan lohkon koodista kaikki testit testaavat. Usein pyritään kattamaan 100% koodista. Jos koodiin ei saada 100% katetta syynä voi olla täysin turhia koodirivejä tai ohjelmoijan ylikatsomia virheitä. Nämä poistamalla voidaan varmistaa, se että niistä ei tule bugeja myöhemmin. Taulukossa 1 esitellään koodin kattavuuden kategorioita, joiden perusteella voidaan jakaa kattavuutta. (Pittet n.d)

Taulukko 1. Koodikattavuuden kategoriat (Pittet n.d)

funktionaalinen	tästä nähdään onko kaikkia funktioita ja aliohjemia kutsuttu
assertiot	tästä nähdään käydyt assertiot
haarat	tästä nähdään kaikki käydyt haarat
konditionaalit	tästä nähdään onko kaikki if- ja else-funktiot käyty
määritelmät	tästä nähdään ovatko kaikki määritelmät käyty
lauseet	tästä nähdään onko kaikki suoritettava koodi käyty läpi
tilat	tästä näkee onko jokaisen vaihdettavan muuttujan tila ollut kaikissa määritetyissä arvoissa

Taulukossa on esitettynä kaikki koodikattavuuden kategoriat. Kaikki kategoriat riippuvat toisistaan, jollakin tavalla mutta ne kaikki ovat silti oma erillinen kategoriansa. (Pittet n.d)

2.4 Muita testaustapoja

Formaali verifikaatio on kollektiivinen termi useammalle eri verifiointitekniikalle. Yhteistä näissä kaikissa tekniikoissa on se, että formaali verifiointi on matemaattinen lähestymistapa. Esimerkiksi predikaattiabstraktiossa luodaan ohjelmasta malli, jonka avulla voidaan selvittää ohjelmasta virhetiloja. (Mäki 2020, 17-18)

2.5 Verifioinnin tärkeysasteet

Koska koodikattavuuden 100 % saaminen maksaa rahaa ja resursseja niin, eikä se aina ole niiden arvoista niin lohko tai ohjelma voidaan todeta verifioiduksi jo 70-80 %. Mitä lähemmäksi kattavuudessa mennään täyttä 100 % niin sitä vaikeampaa viimeisten prosenttien saaminen yleensä on. Aikataulun ja resurssien takia voidaan todeta, että pienempi koodikattavuus riittää. (Cornett 2013) Jos DUT:issa on signaaleja tai muita vastaavia pätkiä, joista ei saada kattavuutta ja joilla ei ole väliä syystä tai toista, voidaan kattavuusohjelmaan tehdä vapautus (waiver), jotta ohjelma ei laske niitä kattavuuteen mukaan.

Vajavainen verifiointi ei kuitenkaan ole riittävä kaikissa tapauksissa. Järjestelmissä, joissa virheillä voi olla paljon vakavampia seuraamuksia, vaatimukset ovat toiset. Esimerkiksi DO-178C-standardi, mikä määrittää ilmailutekniikan turvallisuuskriittisten järjestelmien pätevyyden, vaatii 100 % kattavuuden järjestelmissä, joissa ohjelmavirheellä voi olla katastrofisia vaikutuksia. (IBM 2014)

2.6 Regressiotestaus

Regressiotestaus on testaustapa, jossa käydään tietyt testit läpi useampaan kertaan jollekin ohjelmalle tai lohkolle. Projektin aikana opittiin, että regressiotestit voidaan käynnistää komentosarjalla, jossa voidaan määritellä mitkä testit tehdään, kuinka monta testiä tehdään ja mitä parametrejä regressioon käytetään. Parametreinä regressiossa voi olla satunnaistamisen asetukset, kerätäänkö koodikattavuutta ja kaikille testeille omat asetukset, mikäli sellaisia tarvitaan.

Regressiotestaus oli opinnäytetyötä tehdessä erinomainen keino testata kaikki testit kerralla ja saada koodikattavuus kokonaisesta laitteesta/ohjelmasta. Tämäkään ei kuitenkaan ole täydellistä, sillä regressiotestaus vie paljon aikaa riippuen testattavan projektin koosta ja monimutkaisuudesta. Pienemmissä projekteissa kuten tässä opinnäytetyössä regressiotestit voivat viedä vain 5-10 minuuttia, kun taas isoissa projekteissa se voi kestää jo useampia tunteja tai päiviä.

3 SYSTEMVERILOG

Tässä luvussa esitellään SystemVerilogin perusteita ja ominaisuuksia.

3.1 Perusteet

Laitteistokuvauskieliä kuten SystemVerilog ja VHDL (very high-speed integrated circuit hardware description language) käytetään kuvaamaan laitteiston logiikkaa ja muuttamaan se digitaalisiksi porteiksi, jotta se voidaan muuttaa fyysiseksi piiriksi. Koodin muuttamista logiikkaporteiksi kutsutaan syntesoinniksi. Tässä työssä keskitytään kuitenkin pääasiassa SystemVerilogin verifiointiominaisuuksiin. Ennen SystemVerilogia, käytettiin verifiointiin Verilogia, josta SystemVerilog pohjautuu. Kun laitteet tulivat yhä monimutkaisimmiksi ja Verilogia jatkuvasti laajennettiin niin vuonna 2005 Verilog, kaikkine laajennuksineen standardoitiin SystemVerilogiksi. Joten SystemVerilog on käytännössä laajennettu Verilog eikä erillinen kieli. SystemVerilogia käytetään suurimmaksi osaksi pelkästään verifiointiin sen ominaisuuksien ansiosta, mutta sitä voi käyttää myös laitesuunnitteluun. (Doulos n.d.)

3.2 Ominaisuudet

Verilogin kasvaessa SystemVerilogiksi, siihen on tullut paljon ominaisuuksia, jotka auttavat verifiointi-insinöörejä verifioimaan yhä suurempia ja monimutkaisempia kokonaisuuksia. Yksi tärkeä ominaisuus on olio-ohjelmointi. SystemVerilogin olio-ohjelmointimalli mahdollistaa testipenkin abstraktion. SystemVerilog mahdollistaa myös ohjatun satunnaistestauksen, mikä on tärkeää suurien kokonaisuuksien verifioinnille. (SystemVerilog Tutorial n.d.)

4 UVM

Tässä kappaleessa esitellään UVM:n perusteita ja ominaisuuksia. Tähän tutustutaan, koska UVM on opinnäytetyön keskeisimpiä metodologioita.

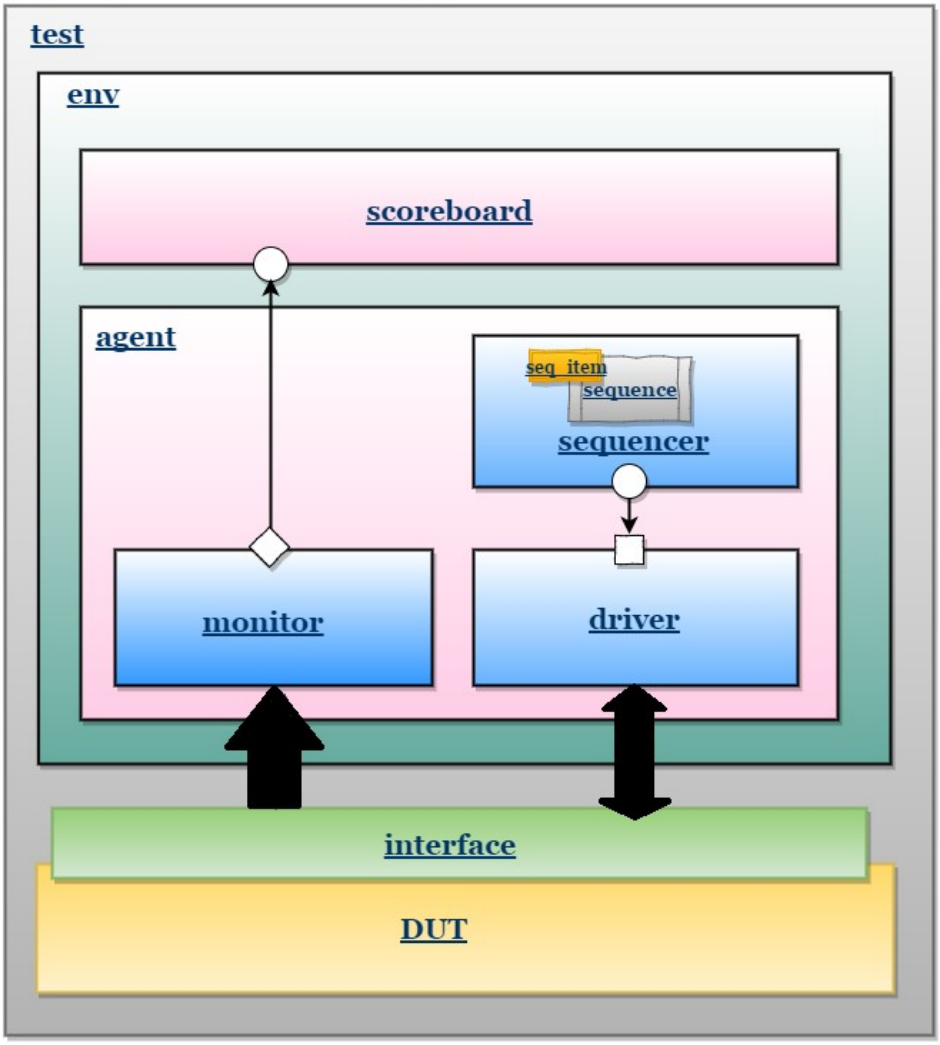
4.1 Perusteet

UVM eli Universal Verification Methodology on standardoitu metodologia verifioimista varten. UVM pohjana on OVM eli Open Verification Methodology, josta UVM ei poikkea paljoa. UVM:n tavoitteena on luoda kokonaisia verifiointiympäristöjä uudelleenkäytettävillä komponenteilla. UVM käyttää omaa kirjastoaan, jossa on paljon automaatiota verifiointia varten. (Aynsley 2010)

UVM:n oppiminen on tunnetusti melko vaikeaa ja aikaa vievää. Pohjana tarvitsee osaamisen SystemVerilogista ja olio-ohjelmoinnista. UVM:n ja SystemVerilogin oppimiseen löytyy paljon materiaalia sekä digitaalisena internetistä, että fyysisenä kirjoista.

4.2 Testipenkin rakenne

UVM testipenkit noudattavat tiettyä rakennetta, mikä tekee komponenttien ja testien uudelleenkäyttämisen helpommaksi.



Kuva 1. Testipenkin rakenne (UVM Environment n.d)

Yllä olevassa kuvassa (Kuva 1) on esitetty UVM-testipenkin rakenne. Tästä nähdään, että testissä, testi on ylimpänä, mikä tarkoittaa, että testi näkee kaiken ja pystyy muokkaamaan kaikkea. Useimmat UVM-testipenkin osista ovat env:in eli ympäristön sisällä. Ympäristön sisällä on agentti ja scoreboard. Monitori ja ajuri toimivat DUT:in kanssa rajapinnan kautta. Monitori on sisääntuloja ja ulostuloja seuraava komponentti, joka voi myös tarkistaa, että laitteiden välinen viestintä on oikeanlaista. Ajuri tietää miten laitteen signaalien pitäisi liikkua ja osaa ajaa niitä oikeaan aikaan oikeaan paikkaan. Agentti sisältää monitorin ajurin ja sekvensserin ja yhdistää kaikki kolme komponenttia. (UVM Environment n.d)

4.3 Sekvenssit

Sekvenssit ovat pääasiallinen keino ohjata syötettä DUT:ille. Sekvenssejä voidaan käyttää esimerkiksi kirjoittamaan ja lukemaan rekistereitä, resetointiin ja ohjaamaan jotain syötettä. sekvensseiden toimintaan kuuluu olennaisesti sequence_item ja sekvensseri. Sequence_item:it ovat sekvenssien sisällä olevia datapaketteja, joilla transaktiot tapahtuvat. Sekvensseri lähettää datan ajurille, joka lähettää sen edelleen DUT:ille.

4.4 Testit

Testit ovat testipenkin hierarkiassa tb_top:in alla, tämä käynnistää testit. tb_top on kaikista ylin komponentti, joka tekee instanssin koko testipenkistä. Testit itse sisältävät kaikki käskyt, joilla testataan piirin toiminnallisuus. Testeistä kutsutaan tarpeellisia sekvenssejä ominaisuuksien ja toimintojen testaamiseen, testit voivat myös säätää muuttujia testin vaatimusten perusteella. Testien määrä riippuu lohkon tai laitteen kompleksisuudesta. Hyvin yksinkertaiseen lohkoon voi riittää vain muutama testi, mutta suureen ja kompleksiin lohkoon voidaan tarvita satoja testejä. Ne kaikki voidaan sitten ajaa regressiolla. (UVM Test [uvm_test] n.d)

4.5 Covergroupit

Covergroup on käyttäjän määrittelemä paketti, jossa määritellään coverpointit, ristikattavuus ja näytteenottohetket. Covergroupista voi tehdä useamman instanssin useampaan eri paikkaan. Coverpoint tai kattavuuspiste määrittää yhden tai useamman seurattavan muuttujan. Seurattavia muuttujia ”katsotaan” joka kerta kun tapahtuu näytteenotto. Näytteenoton voi joko sitoa johonkin kellon hetkeen tai asettaa tapahtumaan käyttäjän haluamaan aikaan. Jos käyttäjä ei halua tietää ihan kaikkia arvoja mitä seurattava arvo on saavuttanut, on coverpointteihin mahdollista määrittää bins -rakenteita, jotka mahdollistavat tiettyjen arvojen seuraamista muuttujassa. Näihin voi asettaa arvoja tai arvoalueita, jotka käyttäjä haluaa mukaan koodikattavuuteen. Näin voidaan helposti nähdä, onko esimerkiksi raja-arvot testattu, tai onko kaikki mahdolliset arvot testattu muuttujista, joista ei normaalisti näkisi mitään simuloinnin aikana. (SystemVerilog Functional Coverage n.d)

Ristikattavuus eli cross coverage on keino määrittää eri kattavuuspisteiden yhdistelmien kattavuutta. Jotta ristikattavuutta voidaan käyttää, pitää ensin tehdä kattavuuspisteet, joille halutaan tehdä ristikattavuus. Yhtä cross-rakennetta voi käyttää vain kahden kattavuuspisteen ristikattavuuden määrittämiseen. Käytännössä ristikattavuudella haetaan sitä, että kaikilla tarvittavilla x-muuttujan arvoilla on saavutettu kaikki y-muuttujan arvot. (SystemVerification Cross Coverage n.d)

```
1   class subscriber extends uvm_subscriber#(item);
2
3   covergroup kattavuusryhma; // covergroup jossa ei ole määritetty näytteenottoa
4     x: coverpoint x { // x on muuttuja jota seurataan
5     // manuaalisesti joka arvolle luodut binit
6       bins zero = {0};
7       bins one = {1};
8     // automaattisesti luodut binit kaikille alueen arvoille
9       bins range[] = {[2:7]};
10
11    y: coverpoint y {
12      bins zero = {0};
13      bins one = {1};
14    // yksi bin johon kaikki muut arvot menevät
15      bins others = default;
16    }
17
18    // crosscoverage x:n ja y:n välille
19    xXy : cross x,y;
20  endgroup
```

Kuva 2. Covergroup esimerkki

Yllä olevassa kuvassa (Kuva 2) on esimerkki kattavuusryhmästä binien kanssa. Tässä voidaan nähdä binien luonti. Tässä esimerkissä on molemmille muuttujille on luotu kolme biniä. Kummallekin on määritelty kaksi arvoa seurattavaksi erikseen. Kun x tai y saavat arvon 0 tai 1, niin se näkyy kattavuusraportissa joko nimellä zero tai one riippuen arvosta. Muuttujalle x on myös määritetty arvojoukko nimellä range. Jos se saa arvoja 2-7, niin ne ilmestyvät raporttiin nimellä range. Muuttujalla y on bin, nimeltä others.Tämä bin sisältää default-arvon mikä tarkoittaa sitä, että jos arvo ei sovi mihinkään muuhun biniin, niin ne menevät sinne. Default-binistä usein näkee tuleeko muuttujassa odottamattomia arvoja, joille ei ole määritetty omaa biniä. Siihen on myös tehty seurattaville x ja y muuttujille ristikattavuus. Tämä tarkoittaa sitä, että jokaisella mukaan lasketulla x:n arvolla on käyty myös jokainen y:n arvo.

5 TESTIEN SUUNNITTELU JA TOTEUTUS

Tässä luvussa kuvaillaan, mitä projektissa on kuvattu lähtötilanne, mitä on tehty ja lopputulos.

5.1 Lähtötilanne

Testipenkki on tehty SystemVerilogilla käyttäen UVM-kirjastoa. Koska työn tekijän kokemus kummastakin oli työn alussa käytännössä olematon, meni työn alussa suurin osa ajasta pelkästään näihin tutustumiseen, ja niiden käyttämisen oppimiseen. Verifioiminenkaan ei ollut tuttua projektin alussa, mikä vaikeutti aloitusta. Koska UVM on tunnettu erittäin jyrkästä oppimiskäyrästä, sen opetteleminen sai prioriteetin heti alussa.

Työ lähti siitä melkein valmiista testipenkistä, johon piti saada nostettua koodikattavuutta ja lisätä muita toimintoja, kuten muutamien sekvenssien lisääminen, kattavuusryhmien lisääminen ja geneeristen parametrien muuttaminen komentoriviltä. Koska testipenkki itse on jo suurimmalta osalta rakennettu ja todettu toimivaksi, testipenkkiin tutustuminen oli helpompaa.

5.2 Sekvenssien teko

Aluksi testipenkissä ei ollut muita sekvenssejä, kuin yksi pakollinen sekvenssi. Testipenkkiä parantaessa päätettiin, että muutama toiminto, jotka olivat toteutettu eri tavalla, muutettiin sekvensseiksi, jotta testipenkistä tulisi selkeämpi ja uudelleenkäytettävämpi. Testipenkkiin oli myös helpompi tehdä muutoksia tämän ansiosta.

Melkein kaikkia sekvenssejä varten oli otettava selvää siitä, kuinka rekisteriarvoja pystyy muokkaamaan RAL:in (Register Abstraction Layer) kautta sekvensseissä. RAL ei aluksi toiminut sekvenssien kanssa. Rekisterinmuokkaamiseen on olemassa erillinen sekvenssi, mikä on tarkoitettu sitä varten. Tällaisesta sekvenssistä esimerkki on seuraavassa kuvassa (Kuva 3).

```
1   class register_sequence extends uvm_reg_sequence#(uvm_sequence #(uvm_reg_item));
2     `uvm_object_utils(reg_config_seq)
3
4     // luokan jäsenet
5     uvm_status_e status;
6
7     extern function
8         new(string name = "reg_config_seq");  // Luokan rakentaja
9     extern virtual task
10        body();
11  endclass : reg_config_seq
12
13  function reg_config_seq::new(string name = "register_sequence");
14    super.new(name);
15  endfunction : new
16
17  task reg_config_seq::body();
18    `uvm_info(get_type_name(),"register configuration sequence body started", UVM_MEDIUM)
19
20    // ota ral databasesta
21    if (!uvm_config_db#(uvm_reg_block)::get(null, "", "ral", model)) begin
22      `uvm_fatal(get_type_name(), "Can't get RAL from config_db")
23    end
24
25    // aseta arvo rekisteriin
26    write_reg( .rg(model.get_reg_by_name("register")), .status(status), .value(5));
27
28    `uvm_info(get_type_name(),"reg_config sequence body ended", UVM_MEDIUM)
29  endtask : body
```

Kuva 3. Rekisterisekvenssin esimerkki

Kuvassa 3 uvm_reg_sequence tai rekisterisekvenssi käyttää UVM:n omaa uvm_reg_item-datapakettia ja write_reg-metodia muuttamaan rekisteriarvoja. Muuten rekisterisekvenssit toimivat samanlailla verrattuna tavallisiin sekvensseihin.

Ensiksi muutettiin sekvenssiksi funktio, joka loi sarjadataa. Tätä käytettiin luomaan testidataa DUT:ille jotta sarjadatan lukemista voidaan testata. Tämä muutos tehtiin suurimmaksi osaksi ainoastaan sen takia että uudelleenkäytöstä tulee helpompaa ja testipenkki selkeytyy. Funktionaalisesti sarjadatan tuotto on yhä lähes identtinen, mutta siitä on vain tehty sekvenssi. Tämä tehtävä toimi myös työn tekijälle ensimmäisenä kunnon kosketuksena UVM:ään ja sen rakenteeseen. Tämän jälkeen tehtiin kaksi sekvenssiä pelkästään aloitusarvojen alustamista varten. Testin alkaessa testipenkin pitää määrittää joihinkin rekisterimuuttujiin aloitusarvoja. Nämä tehtiin aluksi vain kaikkien testien ylimpänä olevassa kantaluokassa. Ratkaisu olisi toiminut hyvin näinkin, mutta uudelleenkäytön ja helppouden takia päätettiin jälleen, että funktio kannattaa muuttaa sekvenssiksi.

Näitä funktioita tehtiin kaksi. Toinen, joka alustaa kaikki rekisterimuuttujat tiedettyihin arvoihin ja toinen, joka alustaa ne satunnaisesti.

5.3 Testien parantaminen

Lähtötilanteessa, mikään testi ei satunnaistanut lähtötilannetta, joten kaikki testit tapahtuivat aina samalla tavalla Alustussekvenssit tulivat hyödyllisiksi tässä. Satunnainen alustussekvenssi laitettiin alustamaan rekisteriarvot aina testin alussa, mutta mikäli testi vaatii testausta tai korjaamista, niin testin voi asettaa ajamaan toisen sekvenssin, joka muuttaa rekisteriarvot tiedetyiksi arvoiksi sen ajaksi. Yksi testi, jonka tarkoitus on testata pääsy kaikkiin RAL:in rekistereihin ei toiminut ja sitä ei ehditty korjata. Testissä käytetään UVM-kirjaston omaa sekvenssiä, joka ei toiminut testipenkissä.

5.4 Regression ajaminen

Lohkon testipenkissä ei ollut regressioita valmiina, joten siihen suurimmalta osalta kokeiltiin, kuinka monta kertaa testit on käytävä läpi, jotta joka testauskerralla saataisiin maksimikattavuus. Koska kyseessä oleva lohko ei ole kovin iso tai kompleksi, ja testejä ei ole paljoa niin regressiotestauksessa ei kestä kauempaa kuin muutama minuutti. Regressiota ei tämän takia kannattanut optimoida kovinkaan paljoa. Mielenkiinnon nimissä regressiota kokeiltiin optimoida laittamalla se ajamaan tietty määrä testejä ja keräämään kattavuus tehdyistä testeistä taulukkoon 2.

Taulukko 2. Regressiosta tehty taulukko

Regressiotaulukko						
testit	1	5	10	20	30	40
total %	81.45	81.69	81.65	81.84	81.84	81.84
line %	91.70	91.70	91.70	92.16	92.16	92.16
branch %	88.00	88.00	88.00	88.57	88.57	88.57
conditional %	72.41	72.41	72.41	72.41	72.41	72.41
expression %	33.33	33.33	33.33	33.33	33.33	33.33
toggle %	86.43	86.43	86.16	86.43	86.43	86.43

Taulukossa 2 on esitetty kuinka monta kertaa testit on simuloitu. Niiden alla on tulokset kokonaiskattavuudesta ja kattavuuden kategorioista prosentteina. Nähdään että prosenttiluvuilla ei ole paljoa eroa keskenään, koska lohko ei ole kovin kompleksi. Tästä voidaan kuitenkin nähdä, että mitä enemmän testejä simuloidaan, sitä paremmiksi tulokset tulevat ja sitä enemmän yhtenäisyyttä tuloksiin tulee. Tässä testipenkissä kattavuus näyttäisi tasoittuvan sen jälkeen, kun jokainen testi käydään 20 kertaa.

5.5 Koodikattavuuden parannus

Koodikattavuus tässä opinnäytetyössä jäi 81,84 %. Koodikattavuudeksi saatua arvoa olisi mahdollista nostaa suuremmaksi hyödyntämällä yleisiä kattavuuden parantamiseen tähtääviä keinoja. Ensimmäisenä keinona on kirjoittaa lisää testejä, jotta saadaan testaamattomia alueita testattua. Voi myös kirjoittaa parempia testejä, jotka mahdollisesti kattavat enemmän. Tai sitten voi poista turhaa koodia, johon ei joko pääse hyvilläkään testeillä tai josta ei ole hyötyä. Tämän projektin tapauksessa koodi oli vanhaa koodia, jota ei haluttu muuttaa, joten vaihtoehdoiksi jää testien kirjoittaminen ja parantaminen.

5.6 Covergroupit

Funktionaalisen kattavuuden kannalta lohkoon päätettiin tehdä covergroup, joka seuraa muutamaa arvoa. Seurattavat muuttujat ovat perusmuuttujia, joita testit testaavat. Ennen covergroupin tekoa, näistä muuttujista ei ollut minkäännäköistä kattavuutta, eikä niiden toimivuudesta voitu olla varmoja.

Covergroupin implementointi lähti sen toimivuuden testaamisesta. Covergroupin voi tehdä oikeastaan mihin tahansa UVM-rakenteessa, kuten esimerkiksi johonkin luokkaan. Koska jokainen testi on itsessään yksittäinen luokka niin testinä tehtiin covergroup testiluokkaan ja kokeiltiin kattavuuden saamista yhdestä muuttujasta. Kun tämä saatiin toimimaan, ongelmana oli se, kuinka saadaan covergroup yhdestä testistä kaikkiin testeihin. Olisi ollut tarpeettoman työlästä luoda oma covergroup jokaiseen testiin yksitellen. Jos sen laittaisi muualle niin covergroup ei ehkä näkisi kaikkia tarvittavia muuttujia. Tämän takia se usein luodaan

uvm_subscriber:in sisällä. Subscriberin analyysiportin ansiosta muuttujien näkyvyyden ei pitäisi olla ongelma, ja testit eivät tarvitse omaan covergrouppia. Tässä tapauksessa kaikki tarvittavat muuttujat olivat RAL:in sisällä, joten sen sisäänrakennetun analyysiportin ansiosta kaikki tarvittavat muuttujat oli mahdollista nähdä. Koska lohkossa ei ollut jo valmiiksi subscriberia, sellainen luotiin kattavuutta varten.

Subscriberissa oleva covergroup saa kerättyä kattavuuden kaikista tarvittavista muuttujista, ja kattavuuden näkee kattavuusraportissa omalta sivultaan. Parannuksena voisi näytteenottamisaikoja säätää niin että kattavuuteen tulisi ainoastaan tarpeelliset näytteet.

5.7 Geneeristen parametrien muuttaminen

Koodikattavuuden ja verifioimisen kannalta todettiin, että geneeristen parametrien vaihtaminen komentoriviltä voi olla hyödyllistä. Se nopeuttaisi simulaatioiden aloittamista ja voi mahdollistaa regression ajamisen eri parametreillä nopeammin.

Geneeriset parametrit määrittävät DUT;in perustoimintaa ja joskus niiden muuttamisesta voi olla hyötyä verifioinnissa. Parametrien muuttamista kokeiltiin ensin UVM:n sisäänrakennetulla komentoriviprosessorilla. Tämä olisi mahdollistanut parametrien muuttamisen testin aloituskomennon yhteydessä omilla argumenteillaan. SystemVerilog ei salli parametrien muuttamista paketin sisällä, missä parametrit olivat, joten päädyttiin ratkaisuun, jossa ratkaisuun, että parametrejä on muutettava testipenkin kääntämiskomennossa. Tämä ei tee parametrien muuttamisesta yhtä nopeaa, kuin jos niitä voisi vaihtaa simulaatiokomennossa, mutta voidaan pitää tarkoitukseensa riittävänä ratkaisuna. Lopuksi parametrien vaihtoa varten muokattiin makefilea, mikä ohjaa testipenkin kääntämistä. Muutosten jälkeen geneerisiä parametrejä voi muuttaa komentoriviltä kääntämisen yhteydessä muutamalla lisäargumentilla, mikä on huomattavasti helpompaa, kuin niiden muuttaminen koodissa.

5.8 Lopputulos

Työn alussa testipenkki oli vain testipenkin luuranko. Melkein kaikki perustoimivuus oli valmiina, mutta testit olivat keskeneräisiä ja toimivuutta tuli lisätä. Koodikattavuus kokonaisuudessaan lohkossa oli pienien muutosten jälkeen, mikä ei ole vielä mitenkään riittävä kattavuus.

Työn lopussa, kun kaikki muutokset on tehty, saatiin kokonaiskattavuudeksi 81,84 %. Koska kattavuus ei ole vielä 100 %, jää verifioijan päätettäväksi onko lohko tarpeeksi verifioitu. Läheskään kaikissa lohkoissa, ei koskaan saada täyttä 100 %, joten ”tarpeeksi hyvän” päättäminen voi olla hankalaa eikä siihen ole yhtenäisiä säätöjä minkä mukaan lohko pitäisi saada verifioitua. Tämä tarkoittaa sitä, että suurimmaksi osaksi päätös jää verifioijan ammattitaidon varaan. Toisaalta kuten luvussa 2.5 todettiin, ohjelmat ja lohko voidaan joskus todeta verifioiduiksi jo 70-80 % ei-kriittisissä applikaatioissa. Koska tämä kyseinen lohko ei vaaranna henkiä tai aiheuta loukkaantumisia mahdollisessa vikatilanteessa niin saavutettu kattavuus voi olla jo tarpeeksi.

6 POHDINTA

Projektin tarkoituksena oli sekä saada testipenkki tehtyä loppuun, että oppia SystemVerilogia sekä UVM:ää. Projekti oli erinomainen oppimisen kannalta. Valmista testipenkkiä tutkimalla ja sitä parantelemalla oppi UVM:n rakennetta ja sekä SystemVerilogin että UVM:n ominaisuuksia. UVM:n oppimista auttoi jo valmiiksi tehty testipenkki, joka oli tehty UVM-metodologiaa seuraamalla. Tästä huolimatta, tulisi tietää jokaisen luokan tarkoitus ja toiminta, jotka eivät ole välttämättä itsestään selviä. Tämän takia verkkosivut olivat erittäin hyödyllisiä tiedon hakemisessa. Etenkin UVM:n rakenteeseen tutustuminen alussa on hyödyllistä, jotta perusluokat tulevat tutuiksi.

Projektissa alun perin oli tarkoitus saada lohko kokonaan verifioiduksi, niin että voidaan sanoa, että se on kokonaan verifioitu. Lohkoon jäi kuitenkin yksi testi, joka jäi toimimattomaksi mikä vaikuttaa koodikattavuuteen. Kattavuusprosenttia ei saatu niin korkeaksi, kuin olisi toivottu. Tästä huolimatta lohko saatiin välttävään kattavuusprosenttiin.

LÄHTEET

Aynsley, J. 2010. UVM Verification primer. Viitattu 13.12.2021. verkkosivu. https://www.doulos.com/knowhow/systemverilog/uvm/uvm-verification-primer/

Chip Verify. n.d. SystemVerilog Tutorial. verkkosivu. Viitattu 9.12.2021. https://www.chipverify.com/systemverilog/systemverilog-tutorial

Chip Verify. n.d. Systemverilog Functional Coverage. verkkosivu. Viitattu 9.12.2021. https://www.chipverify.com/systemverilog/systemverilog-functional-coverage

Chip Verify. n.d. UVM Test [uvm_test]. verkkosivu. Viitattu 9.12.2021. https://www.chipverify.com/systemverilog/systemverilog-functional-coverage

Cornett, S. 2013. Minimum Acceptable Code Coverage. verkkosivu. Viitattu 9.12.2021. https://www.bullseye.com/minimum.html

Doulos, n.d. What is SystemVerilog?. verkkosivu. Viitattu 15.6.2022. https://www.doulos.com/knowhow/systemverilog/what-is-systemverilog/

IBM. 2014. DO-178C compliance: turn an overhead expense into a competitive advantage. Viitattu 17.1.2022. https://www.ibm.com/downloads/cas/E2GGOGY4

Lemiengre, L. 2021. Making sense of HDL Verification Methodologies. verkkosivu. Viitattu 14.12.2021. https://insights.sigasi.com/opinion/verification/verification-overview/

Mäki, T. 2020. Ohjelmien formaali verifiointi. Tietojenkäsittelytiede. Helsingin Yliopisto. Pro gradu -tutkielma. Viitattu 13.6.2022. https://helda.helsinki.fi/bitstream/handle/10138/314278/Maki_Timo_Pro_gradu_2020.pdf?sequence=3&isAllowed=y

Pittet, S. n.d. An introduction to code coverage. verkkosivu. Viitattu 13.12.2021.https://www.atlassian.com/continuous-delivery/software-testing/code-coverage

Verification Guide. n.d. SystemVerification Cross Coverage. verkkosivu. Viitattu 19.12.2021. https://verificationguide.com/systemverilog/systemverilog-cross-coverage/

Verification Guide. n.d. Systemverilog Functional Coverage. verkkosivu. Viitattu 19.12.2021. https://verificationguide.com/systemverilog/systemverilog-functional-coverage/

Verification Guide. n.d. UVM enviroment. verkkosivu. Viitattu 19.12.2021. https://verificationguide.com/uvm/uvm-environment-example/