Tampereen ammattikorkeakoulu

UVM-testipenkin toteuttaminen IP-lohkotasolla

Mirko Lindroth

OPINNÄYTETYÖ
Huhtikuu 2021

Tieto- ja viestintätekniikan tutkinto-ohjelma
Sulautetut järjestelmät ja elektroniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikan tutkinto-ohjelma
Sulautetut järjestelmät ja elektroniikka

LINDROTH, MIRKO:
UVM-testipenkin toteuttaminen IP-lohkotasolla

Opinnäytetyö 100 sivua
Huhtikuu 2021

Opinnäytetyön tavoitteena oli tuottaa yritykselle testipenkki erään IP-lohkon laitteistokuvauksen verifioimiseksi käyttämällä UVM:ää (Universal Verification Methodology). Työtä lähdettiin ensisijaisesti tekemään, koska aihe mahdollisti perehtymisen UVM:ään sekä testipenkkien toteuttamiseen. Lisäksi yrityksestä löytyi IP-lohko, jolla oli tarvetta testipenkille. UVM-testipenkki toteutettiin alusta saakka itsenäisesti, mikä vaati paljon tutkimustyötä UVM:n kantaluokkakirjaston metodeista ja käytöstä. Opinnäytetyön tarkoituksena oli käydä läpi UVM-testipenkin toteuttamiseen liittyvää teoriaa ja esittää sen toteutusprosessia. Työhön liittyi luottamuksellisen sisällön käsittelemistä, joka on joko jätetty opinnäytetyön julkisesta versiosta kokonaan pois tai abstrahoitu tunnistamattomaksi.

Testipenkkiä ei saatu toteutettua kokonaan, mutta opitun perusteella osataan viedä sen toteuttaminen loppuun saakka. Testipenkin toteutuksesta kuvataan kuitenkin kaikki sisältö, joka ei sisällä luottamuksellista tietoa, ja lukijalle pitäisi opinnäytetyön perusteella syntyä mielikuva testipenkkien toteuttamisen prosessista. Työn aikana opittiin, mikä tekee UVM:n käytöstä haastavaa ja sen pohjalta osataan neuvoa tai ohjata tarvittaessa myös muita työntekijöitä helpommin aiheen pariin.

Johtopäätöksenä, voidaan todeta, että UVM:n oppimisen tekee haastavaksi kantaluokkakirjaston käyttäminen metodologisesti. Opitun pohjalta huomattiin, että testipenkkejä voisi olla mahdollista generoida automaattisesti tiettyyn pisteeseen saakka. Haasteena testipenkkien automatisoinnissa on tiettyjen luokkien ainutlaatuisuus ja toimenpiteet, joita ei voida toistaa testipenkistä toiseen.

Asiasanat: UVM, SystemVerilog

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Embedded Systems and Electronics

LINDROTH, MIRKO:
UVM-testbench implementation at the IP Block Level

Bachelor's thesis 100 pages
April 2021

The objective of this thesis was to produce a testbench to verify a certain IP block’s hardware description using the Universal Verification Methodology (UVM). The work was primarily developed to learn the UVM and the implementation of testbenches, and the commissioner of this thesis had an IP block that required a testbench. The testbench was carried out independently from the beginning, necessitating extensive research on the methods and use of the UVM base class library. The purpose of this thesis was to provide the theory needed to implement a UVM testbench and the process behind it. The work involved confidential content that was either left out altogether or abstracted to be unrecognizable.

The testbench was not completed, but based on the knowledge gained from this work, the remaining implementation can be carried out. However, all content that does not contain confidential information about the testbench is described in this thesis, and the reader should get sufficient information of the process of implementing testbenches with UVM. The work on this thesis revealed challenges associated with the use of UVM and other employees can be advised or guided on the topic more efficiently based on this experience if necessary. In conclusion, using the UVM base class library in a methodological way is what makes learning UVM so difficult.

Based on what was learned, there might be a way to automatically generate testbenches up to a certain degree. The challenge in automating testbench implementations would be the uniqueness of certain classes and steps that cannot be replicated from one testbench to another.

Key words: UVM, SystemVerilog

SISÄLLYS

1 JOHDANTO 7
2 RTL-TASON VERIFIOINTI 9
2.1 Päämäärä 9
2.2 Spesifikaatiot 9
2.3 Testipenkki 10
2.4 Simulaattorit 10
2.5 Verifioinnin merkitys ASIC-piireissä 11
3 SYSTEMVERILOG 12
3.1 Standardi 12
3.2 Suosio 12
3.3 Ominaisuuksia 14
3.3.1 Olio-ohjelmointi 14
3.3.2 Kääntäjädirektiivit 17
3.3.3 Paketointi 20
3.3.4 Proseduraalinen ajoituskontrolli 21
3.3.5 Simulaation aikayksikkö ja tarkkuus 22
3.3.6 Prosessit ja proseduurit 23
3.3.7 Funktiot ja tehtävät 26
3.3.8 Silmukat 27
3.3.9 Väylät 28
3.4 Näkyvyysalue 30
3.5 Lohkojen ilmentymien luominen 32
3.6 Käännösjärjestys 33
4 UVM 34
4.1 Päämäärä 34
4.2 UVM-hierarkia 34
4.3 UVM-tietokannat 37
4.4 UVM-tehdas 41
4.5 UVM-vaiheet 42
4.6 TLM-rajapinta 45
4.7 Testipenkkilohko 48
4.8 DUT wrapper 48
4.9 Sekvenssi alkio 49
4.10 Sekvenssi 49
4.11 Sekvensoija 51
4.12 Ajuri 51

4.13 Monitori ... 54
4.14 Agentti ... 55
4.15 Tulostaulu ... 56
4.16 Rekistereiden kanssa vuorovaikuttaminen ... 58
4.16.1 Rekisterilohkot ... 58
4.16.2 Rekisterirajapinta ... 59
4.16.3 Adapteri ja syötteenennustaja ... 61
4.16.4 Rekisterirajapinnan ympäristö ... 62
4.17 Ympäristö ... 63
4.18 Testi ... 64
5 TESTIPENKIN SUUNNITTELU JA TOTEUTUS ... 67
5.1 Suunnittelu ... 67
5.2 Simulaatioympäristön valmistelut ... 68
5.3 Alustava testipenkki ... 69
5.4 Top-taso ... 71
5.4.1 DUT wrapper ... 71
5.4.2 Väylät ... 73
5.4.3 Väylien liittäminen ... 74
5.4.4 Testipenkkilohko ... 75
5.5 Sekvenssit ... 75
5.5.1 Sekvenssi alkio -luokka ... 76
5.5.2 Sekvenssiluokka ... 77
5.6 Sekvenssien ajaminen ... 78
5.6.1 Sekvensoijaluokka ... 79
5.6.2 Ajuriluokka ... 80
5.6.3 Ajurin testaaminen ... 81
5.7 Sekvenssien monitorointi ... 85
5.7.1 Monitoriluokka ... 85
5.7.2 Monitorin testaaminen ... 87
5.8 Rekistereiden lukeminen ja kirjoittaminen ... 88
5.8.1 RAL-ympäristö ... 88
5.8.2 RAL-ympäristön testaaminen ... 90
5.9 Testipenkin viimeistely ... 91
5.9.1 Tulostaulu luokka ... 92
5.9.2 Ympäristöluokka ... 94
6 JOHTOPÄÄTÖKSET JA POHDINTA ... 96
LÄHTEET ... 98

LYHENTEET JA TERMIT

ASIC	Application Specific Integrated Circuit
DUT	Device Under Test
eng.	englanniksi
EDA	Electronic Design Automation
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IP	Intellectual Property
n.d	no date
RAL	Register Access Layer
RTL	Register Transfer Level
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VIP	Verification IP

1 JOHDANTO

Transistoreiden määrä mikropiireissä on kasvanut pitkään Mooren lain (Moore 1965) mukaan, eli noin kaksinkertaisesti joka toinen vuosi. Tämä on mahdollistanut yhä monimutkaisempien mikropiirien valmistamisen. Nykypäivän järjestelmäpiirit sisältävät lukuisia toimintoja ja ovat niin monimutkaisia, että niiden havainnollistaminen sekä suunnittelu jaetaan hierarkkiseen lohkorakenteeseen.

Suunniteltaessa järjestelmäpiirin rakennetta, sen eri toiminnot ryhmitellään osalohkoiksi. Osalohkot, esimerkiksi digitaalista signaalikäsittelyä prosessoiva lohko, saattavat puolestaan sisältää omia osalohkojaan ja niistä muodostuvaa järjestelmäpiirin kokonaisuutta esitetään usein lohkokaavioilla kuvaamaan piirin rakennetta. Järjestelmäpiirin suunnittelun hajottaminen osalohkoihin myös nopeuttaa piirin suunnitteluprosessia sekä sen saattamista markkinoille, kun useampi lohko voidaan suunnitella riippumattomasti toisistaan samanaikaisesti.

Osalohkojen toiminnot kehitetään ASIC (Application Specific Integrated Circuit) -järjestelmäpiireille aina piirikohtaisiksi. Toisinaan osalohko tai sen sisältämä toiminto saattaa olla useasti toistuva tai halutaan käyttää uudelleen myös muiden piirien suunnittelussa. Tällöin koko lohkosta tai sen tietystä toiminnosta luodaan uudelleenkäytettävä osalohko, jota kutsutaan valmislohkoksi tai IP-lohkoksi (eng. Intellectual Property block). IP-lohkot vastaavat toiminnoiltaan ASIC-piirejä, mutta eivät useimmiten ole itsenäisiä moduuleja. Niiden tarkoituksena on mahdollistaa toimintojensa uudelleenkäytettävyys ja toimimaan rakennuskomponentteina monimutkaisempien lohkojen suunnittelussa. Muut osalohkot, jotka saattavat olla myös itsessään IP-lohkoja, voivat sitten hyödyntää IP-lohkoja sisällyttämällä ne suunniteluunsa.

ASIC-piirien tuotekehitysprosessi sisältää suuren määrän eri vaiheita. Suunnitteluprosessi alkaa piirin spesifikaatioilla ja arkkitehtuurin suunnittelusta, jonka jälkeen piiri ja sen osalohkojen suunnittelu toteutetaan RTL-tasolla (Register Transfer Level). RTL-suunnittelu on abstraktinen suunnittelutaso, joka mallintaa synkronisen digitaalipiirin logiikkaa signaalien siirtymisinä rekistereiden välillä. Tänä päivänä RTL-tason suunnittelu toteutetaan laitteistokuvauskielillä, joista syntyy laitteistokuvauksia alempien suunnittelutasojen esitysten luomiseksi. IP-lohkon

laitteistokuvaus on kuitenkin verifioitava ennen sen liittämistä toisen lohkon tai järjestelmäpiirin kuvauksen osaksi. Verifiointia toteutetaan myös monissa muissa suunnitteluprosessin vaiheissa varmistuakseen piirin virheettömyydestä, mutta tässä opinnäytetyössä käydään läpi ainoastaan RTL-tason verifiointia.

Työn tavoitteena oli tuottaa yritykselle testipenkki erään ASIC-järjestelmäpiirin IP-lohkon laitteistokuvauksen verifioimiseksi sekä oppia mikropiirien verifioinnissa yleiseksi tullutta metodologiaa nimeltä UVM (Universal Verification Methodology). Laitteistokuvausten verifiointi toteutetaan simulaattoreiden avulla, jotka ymmärtävät laitteistokuvaus-, laitteistoverifiointi- sekä laitteistokuvaus ja -verifiointikielillä toteutettuja testipenkkejä. Testipenkkien tarkoituksena on antaa syötteitä verifioitavalle lohkolle eli DUT:lle (Device Under Test) stimuloidakseen sitä ja verrata sen ulostuloja odotettuihin arvoihin.

Työn tarkoituksena on opastaa lukija testipenkin toteuttamisen prosessin eri vaiheiden läpi ja esittää teoriaosuudessa ASIC-järjestelmäpiirien IP-lohkojen RTL-tason verifioinnista, verifioinnin merkitystä ASIC-piirien suunnittelussa, UVM:ää sekä testipenkin kirjoittamiseen käytettyä SystemVerilog laitteistokuvaus- ja verifiointikieltä. SystemVerilogin ja UVM:n avainsanat sekä lähdekoodien käskyt erotellaan muusta tekstistä käyttämällä kursiivia.

2 RTL-TASON VERIFIOINTI

Tässä luvussa esitellään RTL-tason verifiointia yleisesti. Luvussa käydään läpi asioita kuten verifioinnin merkitys ASIC-piireille sekä vastataan kysymyksiin kuten, mikä on testipenkki tai, mikä sen tarkoitus on.

2.1 Päämäärä

RTL-tason verifioinnin tarkoituksena on varmistaa lohkon tai piirin laitteistokuvauksen virheettömyys. IP-lohkon RTL-tason verifiointi toteutetaan ennen sen käyttämistä korkeammalla lohkohierarkiassa olevissa lohkoissa, jotta sen sisältämät mahdolliset virheet olisivat korjattu ennen käyttöä. Tämä vähentää ongelmien selvittämiseen menevää aikaa korkeammalla lohkohierarkiassa.

RTL-tason verifiointi on erityisen tärkeää, koska laitteistokuvauksia käytetään tuottamaan matalamman tason kuvauksia. Ilman toimivaa ja verifioitua laitteistokuvausta, sen virheet siirtyvät myös pahimmassa tapauksessa tuotantoon meneville piireille. Valmistettua piiriä ei voida enää muuttaa ja ainoa mahdollisuus korjata piiri jälkikäteen on, jos ohjelmoimalla voidaan muuttaa piirin käyttäytymistä siten, että ongelma pystytään kiertämään.

RTL-tasolla verifioidaan vain ja ainoastaan piirin tai lohkon logiikka, joka esitetään simulaattoreissa ideaalisena. Tämä tarkoittaa, että simulaattorit eivät esitä reaalista fysikaalista maailmaa, jossa esimerkiksi signaalien nousureunat olisivat kaltevia johtuen kapasitanssista tai signaalitasojen muutoksissa saattaisi olla viivettä eri osien välillä reitityksen takia.

2.2 Spesifikaatiot

Ennen piirin verifiointi tuotetaan spesifikaatiodokumentti, jota vasten verifiointi-insinöörit verifioivat piiriä. Spesifikaatiot vastaavat kysymyksiin, mitä testataan ja, miten testataan.

Usein piirille tai lohkolle on luotu myös oma suunnitteludokumentti, joka kuvaa piirin eri toimintoja ja ohjelmoitavia rekistereitä. Tätä suunnitteludokumenttia käytetään yleensä verifiointispesifikaatioiden pohjana ja sieltä pyritään poimimaan kaikki ominaisuudet, jotka tulisi testata.

2.3 Testipenkki

Testipenkkejä voidaan luoda millä tahansa laitteistokuvaus-, laitteistoverifiointi- tai laitteistokuvaus- ja verifiointikielellä. Eri kielillä luodut testipenkit toteutetaan hieman eri tavoin, mutta niiden päämääränä on kaikilla syöttää DUT:lle stimulusta ja verrata sen ulostuloja odotettuihin arovihin. Simulaattori kääntää testipenkin lähdekoodit ja simuloi DUT:n käyttäytymistä simulaation aikana, kun DUT:lle annetaan syötteitä testipenkistä.

2.4 Simulaattorit

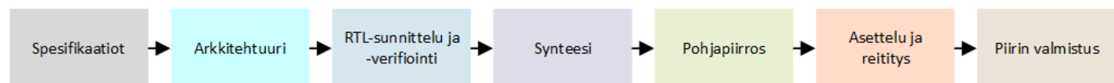
Simulaattorit ovat työkaluja, joilla tutkia DUT:n käyttäytymistä testipenkissä. Simulaattorit ajoittavat jokaisen tapahtumahetken ja seuraavat testipenkin syötteitä. Yleisimpiä simulaattoreita ammattikäyttöön ovat Mentorin Questa (Questa Advanced Simulator n.d.), Synopsyksen VCS (VCS n.d.) sekä Cadencen Xcelium (Xcelium Logic Simulation n.d.). Harrastekäytössä suosituimpina simulaattoreina ovat Questasta riisuttu versio ModelSim (Modelsim n.d.) sekä WEB-pohjainen Eda Playground (EDA playground n.d.).

Simulaattorit eivät pelkästään simuloi DUT:n käyttäytymistä vaan sisältävät myös kääntäjät (eng. compiler) lähdekoodien kääntämiseksi. Useimpia simulaattoreita voidaan käyttää sekä graafisella käyttöliittymällä että syöttämällä simulaattorikomentoja komentoriviltä. Tämä mahdollistaa skriptien kirjoittamisen, jolla helpotetaan sekä automatisoidaan testien ajamista. Simulaattorit sisältävät kääntämisen ja simuloimisen lisäksi paljon muitakin ominaisuuksia, jotka vaihtelevat simulaattorista toiseen. Kaikki simulaattorit kuitenkin sisältävät aaltomuotoeditorin, joka

on välttämätön tutkimaan signaalitasojen muutoksia visuaalisesti. Muita ominaisuuksia saattavat olla esimerkiksi kattavuuden kerääminen sekä pysäytyspisteiden asettaminen testipenkin lähdekoodeihin.

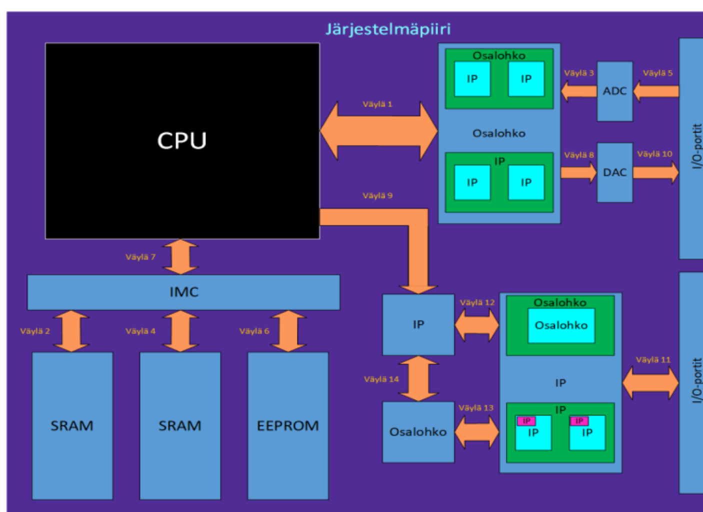
2.5 Verifioinnin merkitys ASIC-piireissä

ASIC-piirien tuotekehitysprosessi (kuvio 1) kestää varsin pitkään ja sisältää useita vaiheita, jotka joudutaan käymään läpi aina, kun uusia virheitä löytyy tai spesifikaatiot muuttuvat. Tuotekehitysprosessin kesto on yleensä useita vuosia ja siksi ASIC-piirien kehitystyö on kallista. Virheiden löytäminen onkin erityisen tärkeää mahdollisimman aikaisessa vaiheessa minimoidakseen kustannukset.



KUVIO 1. ASIC-piirien tuotekehitysprosessi

Verifioinnin merkitys ASIC-piirien tuotekehityksessä on suuri, koska toisin kuin FPGA (Field-Programmable Gate Array) -piirit, ASIC-piirejä (kuvio 2) ei pystytä enää jälkikäteen muuttamaan. Valitettavan usein virheitä löydetään myös vasta pitkäaikaisessa käytössä. Virheitä voi esiintyä myös useissa tuotekehityksen vaiheissa ja pelkkä laitteistokuvauksen verifioiminen ei siis riitä. Ongelmat myöhemmissä vaiheissa saattavat olla myös ainoastaan ratkaistavissa RTL-tasolla, joka tuo lisää kustannuksia ja viivästyttää tuotekehityksen aikataulua.



KUVIO 2. Kuvitteellinen ASIC-piiri

3 SYSTEMVERILOG

Tässä luvussa tutustutaan SystemVerilog laitteistokuvaus ja -verifiointikieleen ja sen ominaisuuksiin. Luvun tarkoituksena on antaa lukijalle jonkinlainen perusymmärrys ja pohja ymmärtääkseen UVM:ää käsitteleviä lukuja.

3.1 Standardi

SystemVerilog on avoin mikropiirien verifioinnissa yleistynyt IEEE:n standardoima laitteistokuvaus ja -verifiointikieli, jonka ensimmäinen standardi IEEE P1800-2005 julkaistiin marraskuussa 2005. Standardissa IEEE 1800-2017 mainitaan SystemVerilogin alun perin pohjautuneen Verilog-laitteistokuvauskieleen, jota Accellera laajensi versiossaan SystemVerilog 3.1a vuonna 2004. Kaikki Verilogin ominaisuudet olivat siis myös käytettävissä SystemVerilogissa.

Myöhemmin standardissa IEEE 1800-2009 yhdistettiin entiset SystemVerilogin sekä Verilogin standardit IEEE 1800-2005 ja IEEE 1364-2005 yhdeksi standardiksi. Tämä loi yhtenäisen laitteistokuvaus- ja verifiointikielen, joka tunnetaan nykyään nimellä SystemVerilog. (IEEE 1800–2017 2018, 8, 39.) Uusin revisioitu versio standardista on IEEE 1800-2017.

3.2 Suosio

RTL-tason suunnitteluun merkittäviä laitteistokuvauskieliä ovat ainoastaan VHDL (Very High-Speed Integrated Circuit Hardware Description Language) sekä SystemVerilog, joka nykyään sisältää entisen Verilogin. Yhtenäinen kieli piirien suunnitteluun ja verifiointiin mahdollistaa sujuvamman kommunikoinnin suunnittelijoiden ja testaajien välillä. Tämän vuoksi Verilogin ja SystemVerilogin standardien yhdistyminen oli tärkeää, mutta myös VHDL:llä on mahdollista suunnitella sekä verifioida piirejä.

EDA-työkalutoimittajat saattavat tukea nykyään paremmin SystemVerilogia kuin VHDL:ää VHDL:n suosion laskun takia tehden työkaluistaan optimoidumpia SystemVerilogille. SystemVerilogin huonona puolena on, että sen standardin työryhmä koostuu EDA-työkalutoimittajien edustajista, jotka saattavat olla haluttomia lisätä ominaisuuksia kielen standardiin, elleivät näe rahallista hyötyä. VHDL:n työryhmä puolestaan muodostuu yksilöllisistä tekijöistä, joten samanlaista ongelmaa ei ole. SystemVerilogin työryhmän koostuminen EDA-työkalutoimittajien edustajista toisaalta varmistaa, että EDA-työkalutoimittajat hyväksyvät ja tulevat tukemaan kielen standardia. (Golson & Clark 2016; Spear 2008, xxxi.)

Suunnittelun näkökulmasta VHDL sekä SystemVerilog ovat kummatkin kieliä, joilla kuvata piirin logiikkaa ja molemmilla voidaan tuottaa laitteistokuvauksia. Tämän vuoksi suunnittelussa käytettävän kielen suosio riippuu paljon siitä, kuinka moni päättää käyttää sitä.

Verifioinnin puolella SystemVerilogin suosiota lisää olio-ohjelmointi sekä UVM. SystemVerilogin olio-ohjelmointi tekee testipenkeistä modulaarisia ja helpottaa huomattavasti testipenkin lähdekoodien ylläpitämistä. Olio-ohjelmoinnin lisäämän suosion lisäksi luotiin UVM, jonka jälkeen verifiointi-insinööreillä oli myös yhtenäinen tapa eli metodologia suunnitella testipenkkejä. Näitä ominaisuuksia ei ole tarjolla VHDL:lle ja suunniteltaessa testipenkkejä VHDL:llä yritetäänkin usein imitoida olio-ohjelmoinnin tekniikoita käyttämällä VHDL-paketteja. Molemmilla kielillä voidaan kuitenkin toteuttaa testipenkkejä, mutta SystemVerilogin ominaisuudet tekevät siitä käytännöllisemmän verifioinnissa.

Tulevaisuudessa, jos EDA-työkalutoimittajat olisivat haluttomia tukemaan tai tarjoaisivat huonompaa tukea VHDL:lle, se tulisi vaikuttamaan merkittävästi käytettävän kielen valintaan. Huonompi tuki lisäisi SystemVerilogin suosiota, koska HDL-kielillä suunniteltavat RTL-tason kuvaukset ovat ainoastaan yhtä hyviä kuin käytössä olevat EDA-työkalut.

3.3 Ominaisuuksia

SystemVerilogin ja UVM:n käytön laaja osaaminen on hyödyksi, mutta myös niiden perusosaamisella pystytään toteuttamaan UVM-testipenkkejä. SystemVerilogin tiettyjen ominaisuuksien ymmärtäminen on kuitenkin välttämätöntä ennen tarkempaa testipenkin toteuttamisen tarkastelua kuten yksi sen tärkeimmistä ominaisuuksista olio-ohjelmointi.

3.3.1 Olio-ohjelmointi

Toisin kuin VHDL, Systemverilog tukee olio-ohjelmointia (eng. object-oriented programming). Olio-ohjelmointia hyödynnetään SystemVerilogissa testipenkkien toteuttamisessa erityisesti jakamalla testipenkit osiin, jotta vältytään kirjoittamasta niiden rakenneosia useampaan kertaan. Esimerkiksi jokainen testi voidaan eritellä omaksi luokakseen (eng. class), jotka sisältävät lähdekoodit DUT:n tietyn ominaisuuden testausta varten. SystemVerilogin olio-ohjelmointi myös mahdollistaa verifiointiympäristöjen uudelleenkäyttämisen, paremman hahmottamisen sekä helpottaa testipenkkien lähdekoodien ylläpitämistä.

Olio-ohjelmoinnissa luodaan luokkia, jotka kuvaavat rakenteiden yhteisiä piirteitä ja sisällytetään yleensä omiin otsikkotaso (eng. header) -tiedostoihin. Olioihin kiinteästi liittyviä muuttujia kutsutaan jäsenmuuttujiksi (eng. data member) ja funktioita (eng. function) sekä tehtäviä (eng. task) yleensä metodeiksi (eng. method). Luokilla kapseloidaan rakenteiden yhteisiä piirteitä abstraktiksi esitykseksi, joita voidaan käyttää muualla lähdekoodeissa luomalla olioita (eng. object). Oliot ovat luokkien dynaamisia ilmentymiä eli luokat toimivat mallipohjina olioiden luomiseksi.

Oliot luodaan SystemVerilogissa kutsumalla *new*-funktiota eli luokan rakentajafunktiota. Olioita ei voida kuitenkaan luoda tietämättä, mistä olion luokasta on kyse. Ensin on luotava muuttuja tyyppiä luokka määrittämään luotavan olion luokan, jonka arvoksi asetetaan olion kahva (eng. object handle). Tyyppiä luokka olevien muuttujien arvot ovat luomisen yhteydessä *null* eli eivät sisällä mitään.

Olion luonti tapahtuu asettamalla tyyppiä luokka olevan muuttujan arvoksi *new*-funktion kutsu, jolloin varataan oliolle muistialue ja säilytetään sen kahva eli olion sijainti muistiavaruudessa. Tämän jälkeen kahva osoittaa luokasta luotuun olioon eli varattuun muistialueeseen. Olion kahvaa käyttämällä päästään muuttamaan luokasta luodun olion jäsenmuuttujia ja käyttämään sen metodeja. Tämä tapahtuu kutsumalla lähdekoodeissa tyyppiä luokka olevan muuttujan nimeä, jonka arvona on olion kahva, ja haluttua jäsenmuuttujaa tai metodia eroteltuna pisteellä (kuva 1).

```
1   class kantaluokka;
2     bit x = 0;
3     function void tulosta;
4       $display("kantaluokan x arvo on: %0d", x);
5     endfunction
6   endclass
7
8   module lohko;
9     kantaluokka olion_kahva;     // luodaan tyyppiä kantaluokka oleva muuttuja
10
11    initial begin
12      olion_kahva = new();      // luo kantaluokasta olio ja säilytä sen kahva
13      olion_kahva.x = 1;        // muuta jäsenmuuttujan arvo
14      olion_kahva.tulosta();    // kutsuu kantaluokan metodia "tulosta"
15    end
16  endmodule
17
# kantaluokan x arvo on: 1
```

KUVA 1. Luokan ja olion luominen sekä jäsenmuuttujan arvon muuttaminen ja metodin kutsuminen

Luokista luotujen olioiden jäsenmuuttujat ja metodit saattavat olla muualla lähdekoodeissa käytettävissä tai eivät, riippuen niiden näkyvyydestä. Luokkien jäsenmuuttujien ja metodien näkyvyydellä tarkoitetaan luokan abstrahointia, jossa peitetään jäsenmuuttujat ja niiden muokkaaminen mahdollistetaan pelkästään metodien avulla. Oletuksena kaikki jäsenmuuttujat SystemVerilogissa ovat tyyppiä *public* eli olioiden kahvojen kautta päästään muokkaamaan jäsenmuuttujia suoraan.

Luokan jäsenmuuttujille voidaan lisätä avainsanat *local* tai *protected* rajoittamaan niiden näkyvyyttä muille lähdekoodeille. Avainsana *local* asettaa näkyvyyden pelkästään luokan sisälle eli vain luokan metodit voivat muokata jäsenmuuttujia ja avainsana *protected* määrää näkyvyyden pelkästään kyseiselle luokalle ja sen aliluokille. Yleensä testipenkeissä ei kuitenkaan peitetä jäsenmuuttujien näkyvyyttä, koska tarkoituksena on saada maksimaalinen kontrolli testipenkistä ja sen sisällä olevista muuttujista ja signaaleista.

Olio-ohjelmointi sisältää myös periytymisen ominaisuus, jonka avulla jo olemassa olevia luokkia voidaan hyödyntää pohjana luomaan uusia luokkia. Periytyvät luokat sisältävät kaikki pohjana käytetyn luokan ominaisuudet ja kutsutaan aliluokiksi. Luokkia, josta aliluokat periytyvät, kutsutaan kantaluokiksi.

Periytyminen tapahtuu SystemVerilogissa luomalla ensin luokka *class*-avainsanalla ja antamalla sille nimi. Kantaluokka, josta luokan halutaan periytyvän, määritetään antamalla kantaluokan nimi avainsanan *extends* perässä. Kantaluokasta ei tarvitse luoda oliota kutsuakseen aliluokasta sen ominaisuuksia. Aliluokka perii kaikki kantaluokan ominaisuudet ja ne käyttäytyvät kuin olisivat määriteltynä aliluokassa kuten kuvassa 2.

```
1   class kantaluokka;
2     bit x = 0;
3     function void tulosta;
4       $display("kantaluokan x arvo on: %0d", x);
5     endfunction
6   endclass
7
8   class aliluokka extends kantaluokka;
9     function void tulosta_2;
10      $display("aliluokassa");
11    endfunction
12  endclass
13
14  module lohko;
15    aliluokka aliluokan_olion_kahva;       // luodaan tyyppiä aliluokka oleva muuttuja
16
17    initial begin
18      aliluokan_olion_kahva = new();       // luo aliluokasta olio ja säilytä sen kahva
19      aliluokan_olion_kahva.tulosta();     // kutsuu kantaluokan metodia "tulosta"
20      aliluokan_olion_kahva.tulosta_2();   // kutsuu aliluokan metodia "tulosta_2"
21    end
22  endmodule
23
# kantaluokan x arvo on: 0
# aliluokassa
```

KUVA 2. Kantaluokan metodien kutsuminen käyttämällä aliluokan oliota

SystemVerilogissa on olemassa avainsana *super*, jonka avulla voidaan myös kutsua kantaluokan metodeja ja antaa uusi arvoja sen jäsenmuuttujille. Kutsumalla aliluokan metodissa *super* ja haluttua kantaluokan jäsenmuuttujaa tai metodia pisteellä eroteltuna (kuva 3), päästään käyttämään kantaluokassa olevia ominaisuuksia myös tapauksissa, jossa aliluokka korvaa kantaluokan määritelmän. (Spear 2008, 125–159, 259–293; Rintala & Jokinen 2005, 54–64, 128–131, 136–148.)

```
1  class kantaluokka;
2    bit x = 0;
3    virtual function void tulosta;  // funktio on virtuaalinen eli voidaan korvata
4      $display("kantaluokan x arvo on: %0d", x);
5    endfunction
6  endclass
7
8  class aliluokka extends kantaluokka;
9    bit x   = 0;
10
11   virtual function void tulosta;
12     super.tulosta();                          // tulostaa kantaluokan x:n arvo on 0
13     super.x = 1;
14     super.tulosta();                          // tulostaa kantaluokan x:n arvo on 1
15     $display("aliluokan x arvo on: %0d", x);  // tulostaa aliluokan x:n arvon
16   endfunction
17 endclass
18
19 module lohko;
20   aliluokka aliluokan_olion_kahva;     // luodaan tyyppiä aliluokka oleva muuttuja
21
22   initial begin
23     aliluokan_olion_kahva = new();     // luo aliluokasta olio ja säilytä sen kahva
24     aliluokan_olion_kahva.tulosta();   // kutsuu aliluokan metodia "tulosta", koska kantaluokan funktio korvataan
25   end
26 endmodule
27
# kantaluokan x arvo on: 0
# kantaluokan x arvo on: 1
# aliluokan x arvo on: 0
```

KUVA 3. Korvatun jäsenmuuttujan ja metodin kutsuminen aliluokasta

Olio-ohjelmoinnin toteutus saattaa erota käytettävästä kielestä toiseen. Kaikille olio-ohjelmointia tukeville kielille pätee kuitenkin tietyt olio-ohjelmoinnin konseptit eli olio-ohjelmoinnin osaaminen yhdellä kielellä riittää usein käyttämään olio-ohjelmointia muillakin kielillä. Ainoastaan tietyt peruskäsitteet olio-ohjelmoinnista esitettiin testipenkin toteuttamisessa käytettävien tapojen ja termien ymmärtämiseksi. Tärkeää olisi ymmärtää luokkien, olioiden kahvojen, jäsenmuuttujien, metodien sekä olioiden merkitykset ja määritelmät olio-ohjelmoinnissa.

3.3.2 Kääntäjädirektiivit

Kääntäjädirektiivit ovat lähdekoodeihin sisällytettäviä lauseita, jotka esikääntäjä tulkitsee ennen varsinaista lähdekoodien kääntämistä. Avataan hieman asiaa ensin Alfred Ahon, Monica Lamin ja Ravi Sethin kirjoittamasta Compilers kirjasta, jossa he selittävät, mikä on kääntäjä. Kääntäjä on yksinkertaisesti ohjelma, joka lukee kirjoitettua lähdekoodia jollakin lähdekielellä ja kääntää sen sitä vastaavaksi halutuksi kieleksi eli kohdekieleksi.

Kääntäjän lisäksi on monia muita ohjelmia, joita saatetaan tarvita luomaan suoritettava ohjelmatiedosto kuten esikääntäjä. Esikääntäjä on siis kääntäjästä erillinen ohjelma, jonka tehtävänä saattaa olla kerätä lähdekoodit niiden ollessa hajautettuna useampaan moduuliin ja tiedostoon tai laajentaa lyhenteitä eli makroja lähdekielen lauseiksi. (Aho, A., Lam, M., Sethi, R. & Ullman, J. 2007, 1–3.)

SystemVerilogin kääntäjädirektiivit astuvat voimaan ilmestyessään ensimmäisen kerran lähdekoodeissa ja alkavat graviksella (`), mutta eivät lopu kaksoispilkkuun. Kääntäjädirektiivit jatkuvat, kunnes toinen kääntäjädirektiivi korvaa edellisen tai viimeinen lähdekooditiedosto on prosessoitu. Kääntäjädirektiivit, eivät myöskään ole sidottuina moduuleihin tai tiedostoihin. (IEEE 1800-2017 2018, 674–689.)

SystemVerilog sisältää yhteensä 22 erilaista kääntäjädirektiiviä, mutta niistä eniten käytettyjä ovat ehtolausekääntäjädirektiivit, *`include*, *`define* ja *`timescale*. SystemVerilog on ottanut paljon vaikutteita C/C++-ohjelmointikielistä, joten useammat sen kääntäjädirektiiveistä jäljentävät kyseisten kielien kääntäjädirektiivejä ja niiden mekanismeja kuten header guard -mekanismia. Ennen header guard -mekanismin tarkastelua on hyvä ymmärtää, että *`include*-kääntäjädirektiivi sisällyttää eli kopioi sen kutsun perään nimetyn lähdekooditiedoston koko sisällön osaksi toista lähdekooditiedostoa sillä rivillä, mistä sitä on kutsuttu.

Verkkosivulla (Header guards 2016) muistutetaan, että muuttujia tai funktioita ei voida määrittää kuin kerran samaan lähdekoodiin. Muuttujien tai funktioiden määrittäminen useammin kuin kerran aiheuttaa käännöksessä virheilmoituksen, jonka voisi korjata poistamalla toistuvat määritykset. Usein kuitenkin päädytään sisällyttämään header-tiedostoja toisten header-tiedostojen osaksi *`include*-kääntäjädirektiivillä, jolloin sisällytetyn header-tiedoston määritys saatetaan sisällyttää useammin kuin kerran.

Erikseen tarkasteltuna tiedostot saattavat olla kunnossa, mutta käännöstulokseen ilmestyy virheilmoituksia useammasta määrityksestä kääntäessä kaikki lähdekooditiedostot yhdessä. Virheilmoitukset (kuva 4) johtuvat lähdekoodin sisällyttäessä `*include*-kääntäjädirektiivillä jonkin määrityksen sisältävän header-tiedoston sekä toisen header-tiedoston, joka sisällyttää itsekin saman määrityksen sisältävän header-tiedoston. Tämä aiheuttaa käännöksessä määrityksen alustamisen kahteen kertaan.

```
1  class luokka_2;
2    bit x = 0;
3
4    function void tulosta;
5      $display("luokassa 2");
6    endfunction
7  endclass : luokka_2
8
9
10
```

```
1  `include "luokka_2.svh"
2  class luokka_1;
3    bit x = 0;
4    luokka_2 a = new();  // luo luokka_2 olio
5
6    function void tulosta;
7      $display("luokassa 1");
8      a.tulosta();
9    endfunction
10 endclass : luokka_1
11
```

```
1  `include "luokka_1.svh"
2  `include "luokka_2.svh"
3  module lohko;
4    luokka_1 b = new();  // luo luokka_1 olio
5
6    initial begin
7      $display("lohkossa");
8      b.tulosta();
9    end
10 endmodule
11
```

```
luokka_2.svh(1): Typedef 'luokka_2' multiply defined.
```

KUVA 4. Määrityksen alustaminen useampaan kertaan

Header guard on kääntäjädirektiiveillä suoritettava toiminto, joka estää toistuvat määritykset samassa lähdekooditiedostossa. Header guard -mekanismi tarkistaa ensin `*ifndef*-kääntäjädirektiivillä, onko jokin makro mielivaltaisella nimellä olemassa ja määrittää sen `*define*-kääntäjädirektiivillä, jos ei ole olemassa. (Header guards 2016.) Kuvassa 5 nähdään edellä esitettyjen lähdekoodien käännöksessä esiintyneen käännösvirheen korjaaminen Head guard -mekanismilla.

```
`ifndef LUOKKA_2_SVH
`define LUOKKA_2_SVH

class luokka_2;
  bit x = 0;

  function void tulosta;
    $display("luokassa 2");
  endfunction
endclass : luokka_2

`endif  // LUOKKA_2_SVH
```

```
`include "luokka_2.svh"
class luokka_1;
  bit x = 0;
  luokka_2 a = new();  // luo luokka_2 olio

  function void tulosta;
    $display("luokassa 1");
    a.tulosta();
  endfunction
endclass : luokka_1
```

```
`include "luokka_1.svh"
`include "luokka_2.svh"
module lohko;
  luokka_1 b = new();  // luo luokka_1 olio

  initial begin
    $display("lohkossa");
    b.tulosta();
  end
endmodule
```

```
# lohkossa
# luokassa 1
# luokassa 2
```

KUVA 5. Header guard -mekanismin käyttäminen

3.3.3 Paketointi

Paketointi on SystemVerilogissa mekanismi jakamaan dataa useammalle moduulille kuten muuttujia, metodeja tai kääntäjädirektiivejä. Pakettien data ei saa sisältää hierarkkisia referenssejä eli ne eivät voi viitata pakettien ulkopuolelle, mutta voivat viitata kyseisen paketin tai muiden tuotujen pakettien sisällä oleviin esineisiin.

Paketeilla on eksplisiittinen näkyvyysalue ja käsitellään samalla tasolla kuin korkeimmalla hierarkiassa oleva moduuli eli top-tasolla. Pakettien esineet tehdään käytettäviksi muille lähdekoodeille tuomalla halutut paketit ja niiden sisältö *import*-avainsanalla. Koko paketin sisältö voidaan tehdä käytettäväksi käskyllä *import::** ja ainoastaan tiettyjä osia spesifioimalla kaksoispisteiden perään haluttu esine. (SystemVerilog package n.d.) Paketteja käytetään usein tuomaan muiden lähdekoodien näkyvyysalueelle samaan kategoriaan kuuluvia lähdekoodeja *`include*-kääntäjädirektiiveillä (kuva 6).

```
2  package seq_pkg;
3      `include "uvm_macros.svh"
4      import uvm_pkg::*;
5
6      `include "seq_item.svh"
7      `include "seq.svh"
8  endpackage : seq_pkg
9
```

KUVA 6. Lähdekoodien jakaminen pakettien avulla

Tiedostojen lähdekoodeja voidaan siis tuoda joko suoraan kääntäjädirektiivillä *`include* tai tuomalla *`include*-kääntäjädirektiivejä sisältävän paketin avainsanalla *import*. Niiden erona on, että *`include* kopioi halutun lähdekoodin kokonaan riville, josta sitä kutsutaan, kun *import* puolestaan tuo ainoastaan paketin sisältämät *`include*-kääntäjädirektiiveillä sisällytetyt lähdekoodit näkyville ja käytettäviksi, mutta kopiointia ei tapahdu.

Lähdekoodien jakamiseksi suositaankin pakettien tuomista *`include*-kääntäjädirektiivin suoran käytön sijan, mutta paketteja ei voida tuoda luokan sisällä. Yleinen menetelmä jakamaan tarvittavat lähdekoodit on luoda paketti samaan kategoriaan kuuluville tiedostoille ja tuoda paketin sisällä muita tarvittavia paketteja.

3.3.4 Proseduraalinen ajoituskontrolli

SystemVerilogilla on kaksi eri ajoituskontrollin tyyppiä: viivekontrolli ja tapahtumalausekkeet. SystemVerilogilla voidaan luoda viiveitä #-, @- ja *wait*-avainsanalla. Erona avainsanojen välillä on, kuinka ne luovat viivettä.

Viivekontrollit ovat yksinkertaisia avainsanan # kutsuja, jonka perään määritellään viiveen määrä simulointiajan aikayksiköissä. Esimerkiksi *#10* toteuttaisi 10 simulointiajan aikayksikön verran viivettä kutsustaan ennen seuraavan käskyn suoritusta.

Avainsanoilla @ tai *wait* voidaan luoda tapahtumakontrollilausekkeita, jotka luovat viivettä seuraavan käskyn suorittamiseen saakka samalla tavalla kuin #-käskyllä. Erona on, että tapahtumakontrollilausekkeille annetaan myös parametriksi odotettavia tapahtumia. Tapahtumana voi olla arvon muutos tai nimenomainen

arvon muutos muuttujassa tai signaalissa. Tapahtumakontrollilausekkeen parametreina voi olla joko yksi tai useampia tapahtumia, joita odottaa. Useat tapahtumat annetaan erottelemalla ne *or*-avainsanalla tai pilkulla.

Avainsanaa @ käytetään myös proseduurien yhteydessä ja sille voidaan määrittää reuna, joilla laukaista ehto. Reunojen tunnistamiseen on käytössä kolme eri SystemVerilogin avainsanaa: *posedge* positiivisille reunoille, *negedge* negatiivisille reunoille ja *edge*, kun halutaan havaita pelkästään jompikumpi reuna.

Reunojen tunnistaminen määritellään suunnanmuutoksiksi arvojen nolla ja yksi välillä. Negatiivinen reuna tunnistetaan muutoksena ykkösestä arvoon *X, Z* tai nollaan sekä arvosta *X* tai *Z* arvoon nolla, jossa *X* kuvaa tuntematonta arvoa ja *Z* kuvaa korkeaa impedanssia. Positiivisen reunan muutos tunnistetaan arvosta nolla arvoon *X, Z* tai yksi sekä arvosta *X* tai *Z* arvoon yksi. Avainsanalla *edge* puolestaan tunnistetaan reuna, kun positiivinen reuna *posedge* tai negatiivinen reuna *negedge* tunnistetaan.

Avainsanalla *wait* voidaan luoda viivettä samoin kuin @-käskyllä, kunnes annetusta ehdosta tulee tosi. Erona @-tapahtumakontrolliin on, että *wait*-käsky odottaa argumenteiksi annettuja tapahtumien tilanmuutoksia, mutta sen kanssa ei voida käyttää tapahtumaehtojen reunojen tunnistusta. (IEEE 1800-2017 2018, 215–223.)

3.3.5 Simulaation aikayksikkö ja tarkkuus

SystemVerilogilla voidaan ajoittaa testipenkin suorituksen etenemistä simulaattorissa eräillä käskyillä, mutta simulaattorille on ilmoitettava ensin, kuinka aika on määriteltynä. Simulaattorille on kerrottava käytettävä aikayksikkö sekä tarkkuus, joka voi olla kaikilla moduuleilla erilainen. Aikayksikkö ja tarkkuus on annettava, jotta simulaattori ymmärtäisi esimerkiksi käskyillä *#10* tai *#0.55* luotuja viiveitä. Aikayksikköä käytetään yleistämään moduulissa tai luokassa käytettävää simulaatioajan aikayksikköä. Aikayksikön sijan voidaan myös tarkentaa viiveille käytettävä aikayksikkö antamalla esimerkiksi käsky *#10ns*. Tämä vaatii kuitenkin, että tarkennettu aikayksikkö on yhtä suuri tai pienempi kuin asetettu aikayksikkö.

Esimerkiksi viiveen *#0.49* simulointiaika on viiveen arvo eli 0,49 kertaa asetettu aikayksikkö. Tämä tarkoittaa, että 10 ns aikayksiköllä saadaan tulokseksi 4,9 ns viive. Tarkkuus puolestaan määrää tarkkuuden asteen viiveille. Tarkkuus 100 ps vastaa 0,1 ns tarkkuutta eli edellisen esimerkin mukaan viive olisi 4,9 ns, mutta tarkkuudella 1 ns saataisiinkin 5 ns. Tämä tarkoittaa, että viiveiden arvot pyöristetään, jos tarkkuus on asetettu liian pieneksi.

Simulaatioiden käyttämä aikayksikkö voidaan asettaa *timeunit*- ja tarkkuus *timeprecision*-avainsanalla tai molemmat voidaan samanaikaisesti asettaa kääntäjädirektiivillä *`timescale*. SystemVerilog tukee yhteensä kuutta eri yksikköä, joiden suuruusluokka voi olla 1, 10 tai 100. Käytettävät yksiköt aikayksikölle, tarkkuudelle ja *`timescale*-kääntäjädirektiiville ovat taulukoituna taulukossa 1. Aikayksikkö voidaan asettaa olemaan 100 sekunnista aina yhteen femtosekuntiin. (IEEE 1800-2017 2018, 55–58; Spear 2008; Verilog Timescale n.d.)

TAULUKKO 1. Aikayksikön ja tarkkuuden mahdolliset yksiköt (IEEE 1800-2017 2018, 55, muokattu)

Merkkijono	Mittayksikkö	arvo sekunneissa
s	sekunnit	1
ms	millisekunnit	1e-3
us	mikrosekunnit	1e-6
ns	nanosekunnit	1e-9
ps	pikosekunnit	1e-12
fs	femtosekunnit	1e-15

3.3.6 Prosessit ja proseduurit

SystemVerilogissa on olemassa useita proseduureja ja niiden luomiseksi käytettäviä avainsanoja. Niiden joukkoon kuuluvat myös funktiot ja tehtävät, jotka esitellään erikseen seuraavassa luvussa. Standardissa IEEE 1800-2017 (2018) esitetään SystemVerilogin proseduurien *initial, always, always_comb, always_latch, always_ff* ja *final* käyttöä ja tarkoitusta, joita käydään läpi tässä luvussa.

Alustavaa proseduuria *initial* käytetään alustamaan tarvittavat asiat kuten muuttujien arvot. Proseduurin *initial* suorittaminen on loputtava ensimmäisen suorituksen jälkeen ja suoritetaan aina simulaation alussa, mutta *initial*-proseduureja voi olla niin monta kuin on tarpeellista.

Proseduurista *always* on olemassa useampia muotoja. Kaikki muodot, mukaan lukien sen perusmuoto *always*, suoritetaan toistuvasti koko simulaation ajan, kunnes kertasuoritettavat *final*-proseduurit astuvat voimaan simulaation lopussa. Proseduurin *initial* tapaan, *always* aloitetaan aina simulaation alussa.

Proseduurin *always* perusmuoto on yleiskäyttöinen proseduuri, jolla toistetaan jatkuvasti jotain toimintoa. Proseduuria voidaan käyttää ainoastaan liitettynä kontrolloituun ajoitukseen muuten, sen silmukkamaisuuden takia, proseduuri ajautuu solmuun.

SystemVerilog tarjoaa proseduurista *always* muunnellun proseduurin *always_comb*, jota käytetään mallintamaan kombinatorisen logiikan käytöstä. Proseduuri käynnistetään ainoastaan kerran ajassa nolla *initial* ja *always* proseduurien jälkeen. Se sisältää automaattisen herkkyyslistan sen lohkon sisällä operaattoreiden oikealla puolella kutsuttavista muuttujista sekä sen sisällä kutsuttavien funktioiden muuttujista. EDA-työkalujen tulisi varmistaa, että *always_comb* suorittaa ainoastaan kombinatorista logiikkaa, eikä sisällä salpoja (eng. latch).

Proseduuri *always_latch* toteuttaa samanlaisen toiminnon kuin *always_comb*, mutta niiden erona on, kuinka EDA-työkalujen tulisi varoittaa niistä. Työkalujen tulisi varoittaa, jos *always_latch*-lohko sisältää muutakin kuin salpoja kuvaavaa logiikkaa.
Proseduuri *always_ff* puolestaan on syntesoitava mallinnus sekvenssisestä logiikasta. Samoin kuin proseduurit *always_comb* ja *always_latch*, EDA-työkalujen tulisi varmistaa *always_ff*-proseduurin oikeanlainen toiminta. Proseduuri *always_ff* ei saa sisältää muuta kuin sekvenssisen logiikan mallinnusta. (IEEE 1800-2017 2018, 205–209.)

SystemVerilogissa voidaan luoda proseduurien lisäksi prosesseja, jotka suorittavat käskyjä rinnakkaisesti simulointiaikaan nähden. SystemVerilogin rinnakkaiset prosessit eivät siis luo oikeita rinnakkaisia prosesseja. Standardissa IEEE 1800-2017 (2018) käydään läpi, kuinka rinnakkaisia prosesseja luodaan ja, kuinka niitä hallitaan. Toisin kuin proseduurit ja sekvenssiset lohkot, jotka alkavat *begin* ja loppuvat *end* avainsanoilla, avainsana *fork* mahdollistaa rinnakkaisten tapahtumien luomisen lohkonsa sisään. Käskyn *fork* luoma lohko päätetään *join, join_any* tai *join_none* käskyillä.

Tapahtumien ei tarvitse olla järjestettynä sekvenssisesti *begin-end*-lohkolla *fork*-lohkossa, mutta voivat olla. Tämä tarkoittaa, että *fork*- lohkon sisällä voidaan luoda sekvenssisiä tapahtumia ja useat *begin-end*-lohkot suorittavat sekvenssisesti käskyjä, mutta rinnakkaisesti toisiinsa nähden. Uusia *fork*-lohkoja voidaan myös luoda toisten *fork*-lohkojen sisällä.

Avainsanat *join, join_any* ja *join_none* määräävät, koska rinnakkaiset prosessit ja tapahtumat poistuvat *fork*-lohkosta. (IEEE 1800-2017 2018, 209–215.) Taulukossa 2 on selitettynä, miten eri *join*-käskyt muuttavat lohkon käyttäytymistä.

TAULUKKO 2. *fork-join* kontrollin valinnat (IEEE 1800-2017 2018, 211)

Valinta	Kuvaus
join	Isäntäprosessi (eng. master process) estää suorituksen jatkamista, kunnes kaikki tämän *fork-kutsun* sisällä luodut prosessit ovat päättyneet.
join_any	Isäntäprosessi estää suorituksen jatkamista, kunnes jokin tämän *fork*-kutsun sisällä luoduista prosesseista päättyy.
join_none	Isäntäprosessi jatkaa rinnakkaista suoritustaan kaikkien tämän *fork*-kutsun sisällä luotujen prosessien kanssa. Luodut prosessit eivät ala suorittamaan ennen kuin isäntäsäie (eng. master thread) suorittaa estokäskyn tai lopettaa suorituksensa.

3.3.7 Funktiot ja tehtävät

SystemVerilogissa on funktioiden (eng. function) lisäksi käytettävissä tehtäviä (eng. task). Molempia käytetään muodostamaan uudelleenkäytettäviä rutiineja sekä jakamaan lähdekoodeja pienempiin osiin, mutta niiden välillä on eroja.

Suurin ero funktioiden ja tehtävien välillä on, että tehtävät voivat sisältää simulointiaikaa kuluttavia käskyjä kuten *#10* eli 10 aikayksikön viive. Funktiot puolestaan eivät voi sisältää mitään simulointiaikaa kuluttavia käskyjä. Funktioiden suorittaminen simulointiaikaan nähden tapahtuvat heti eli yhdessä simulointiaikayksikössä, eivätkä siis askella simulointiaikaa. Funktiot sekä tehtävät suorittavat sisältämänsä käskyt aina sekvenssisesti, jopa suorittaessa niitä rinnakkaisina prosesseina. (IEEE 1800-2017 2018, 319–330.) Taulukossa 3 esitetään funktioiden ja tehtävien eroja.

TAULUKKO 3. Funktioiden ja tehtävien erot (Verilog Task n.d., muokattu)

Funktiot	Tehtävät
Eivät voi sisältää aikaa kuluttavia käskyjä tai viiveitä ja suoritetaan yhdessä simulaatioaikayksikössä	Voivat sisältää aikaa kuluttavia lauseita tai viiveitä eli tehtävien suoritus saattaa loppua muussa simulointiajassa kuin aloittaessa
Eivät voi kutsua tehtäviä sisällään yllä mainitun syyn vuoksi, mutta voivat kutsua muita funktioita	Voivat kutsua sisällään muita funktioita sekä tehtäviä
Eivät palauta arvoa funktion ollessa tyyppiä *void*, muussa tapauksessa täytyy aina palauttaa arvo	Eivät voi palauttaa arvoja, mutta voivat muokata ulostulojen arvoja
Argumenteiksi voidaan antaa tietotyyppejä ilman suuntaa tai määrätyllä suunnalla *input, output, inout* tai *ref*	Argumenteiksi annettavilla tietotyypeillä on aina oltava suunta *input, output, inout* tai *ref* ja määräämättömissä tapauksissa on vakiona *input*

3.3.8 Silmukat

SystemVerilogissa on määritettynä kuusi erityyppistä silmukkarakennetta. Kaikki silmukat eroavat käytökseltään, mutta jokainen niistä toteuttaa saman tarkoituksen. Silmukoiden tarkoitus on suorittaa niiden sisällä annetut käskyt, kunnes jokin ehto on tosi. SystemVerilogin sisältää silmukat

- *repeat*
- *while*
- *for*
- *do while*
- *foreach*
- *forever*.

Silmukka *for* suorittaa käskyjä tietyn kierrosmäärän verran. Silmukan parametreiksi luodaan jokin muuttuja, jonka nimi sekä tyyppi on usein kokonaisluku *i*. Muuttujan arvoon lisätään tai siitä vähennetään yksi jokaisella kierrokselle ja käytetään vertailukohteena ehdolle. Esimerkkinä käsky *for (int i = 0; i < 10; i++)* suorittaisi käskyjä 10 kertaa, jos muuttujan *i* arvoa ei muuteta.

Silmukka, jolle voidaan antaa numeerinen arvo kierrosten suoritusmääräksi, on *repeat*-silmukka. Arvon ollessa tuntematon (*X)* tai korkea impedanssinen (*Z)*, annetun argumentin arvoa kohdellaan kuin se olisi nolla, jolloin poistutaan silmukasta. Esimerkiksi *repeat (10)* käsky suorittaisi käskyt kymmenen kertaa.

Helpottaakseen taulukoiden alkioiden läpikäyminen, SystemVerilog sisältää *foreach*-silmukan. Silmukalle annetaan argumentiksi useita alkioita sisältävä muuttuja, jonka alkioiden mukaan luodaan silmukan iteraatiokerrat. Muuttuja voi olla myös moniulotteinen. Silmukan argumenttina olevaan muuttujan hakasulkeiden sisään annetaan pilkulla erotettuna niin monta muuttujaa kuin on ulottuvuuksia (kuva 7).

```
1   module lohko;
2     int taulukko[1:0][1:0] = '{'{1,2},'{3,4}};;
3
4     initial begin
5       foreach(taulukko[x,y]) begin
6         $display("x=%0d, y=%0d arvo=%0d",x,y, taulukko[x][y]);
7       end
8     end
9   endmodule
10
# x=1, y=1 arvo=1
# x=1, y=0 arvo=2
# x=0, y=1 arvo=3
# x=0, y=0 arvo=4
```

KUVA 7. *foreach*-silmukan toiminta

Silmukka *while* suorittaa puolestaan jatkuvasti käskyjä, kunhan argumenttina oleva ehto on tosi. Silmukan suorittaminen loppuu, kun ehto on epätosi. Silmukassa määriteltyjä käskyjä ei suoriteta lainkaan, jos ehto oli epätosi jo silmukkaan mentäessä.

Tarvittaessa *while*-silmukkaa, joka suorittaisi vähintään kerran käskyt, käytetään *do while* -silmukkaa. Silmukka *do while* suorittaa käskyt vähintään kerran, vaikka ehto olisi epätosi jo silmukkaan mentäessä.
Viimeisenä silmukkana on *forever*-silmukka, joka suorittaa nimensä mukaisesti ikuisesti käskyjä. Välttääkseen ikuista silmukkaa, joka jättäisi simulaation ikuisesti suorittamaan vain tiettyjä käskyjä, silmukkaa tulisi aina käyttää joidenkin ajoituskontrollikäskyjen kanssa. (IEEE 1800-2017 2018,313–317.)

Luokissa ei voida käyttää proseduuria *always*, mutta *forever*-silmukka on sallittu. Proseduurin *always* toimintaa voidaan mallintaa luokissa *forever*-silmukoilla ja *fork*-lohkoilla.

3.3.9 Väylät

SystemVerilog tuo mukanaan helpon tavan ryhmitellä signaaleita käyttämällä väylän käsitettä. Väylän rakenne luodaan avainsanalla *interface* ja antamalla väylälle nimi. Väylä on SystemVerilogissa rakennelohko, jonka sisään määritellään väylän yksittäiset signaalit sekä mahdolliset funktiot tai tehtävät signaalien pro-

sessointia varten. Kaikki signaalit voidaan liittää helposti lohkosta toiseen käyttämällä väylää lähdekoodeissa sekä uudelleen käyttää päivittämällä väylää tarpeen mukaan rikkomatta jo olemassa olevia rakenteita.

Väylää käytetään lähdekoodeissa luomalla ensin ilmentymä kyseisestä väylästä. Väylästä luodaan uusi ilmentymä (kuva 8) kutsumalla väylän nimeä, josta halutaan luoda ilmentymä ja sen perässä annetaan ilmentymällä oma nimi tunnisteeksi. (IEEE 1800-2017 2018, 49–50; SystemVerilog Interface n.d.)

```
1  interface common_if;
2      logic t = 1, reset=0, dout=0;
3       bit clk = 1;
4
5    initial begin : start_clock
6      forever begin : run_clock
7        #20;
8        clk = ~clk;
9      end : run_clock
10   end : start_clock
11 endinterface : common_if
12
13
1  module dut_top;
2    `include "uvm_macros.svh"
3    import uvm_pkg::*;
4
5      // Luo väylän ilmentymä
6      common_if dut_if();
7
8      // Luo ip lohkosta ilmentymä nimeltä dut ja liitä väylän signaalit
9      ip_lohko dut (
10         .t(dut_if.t),
11         .clk(dut_if.clk),
12         .reset(dut_if.reset),
13         .dout(dut_if.dout));
14 endmodule : dut_top
15
```

KUVA 8. Väylä ja väylän ilmentymän luominen lohkon sisään

Ilmentymän nimen perässä luomisen yhteydessä on asetettava aina lopuksi sulkumerkit perään, vaikka väylä ei sisältäisikään argumentteja. Väylän sisältämiä signaaleita kiinnitetään lohkojen portteihin tai signaaleihin antamalla argumentiksi tai arvoksi ilmentymän nimi ja sen perässä haluttu signaali tai metodi eroteltuna pisteellä. Tämän takia väylä ja sen käyttö saattaa muistuttaakin kovasti luokkien ja olioiden käyttöä. Itse luokissa ei voi luoda väylistä ilmentymiä olioiden dynaamisuuden vuoksi, mutta virtuaaliset väylät ovat puolestaan sallittuja.

Virtuaalinen väylä on osoitin alkuperäiseen väylään. Muutokset virtuaalisen väylän signaaleihin heijastuvat suoraan väylällekin, joten käyttämällä virtuaalisia väyliä voidaan muuttaa väylien signaalien arvoja myös olioilla. (IEEE 1800-2017 2018, 748–774.) Testipenkki ja DUT liitetäänkin luomalla DUT:lle väylän ilmentymä ja testipenkille virtuaalinen väylän ilmentymä, jonka avulla näiden kahden lohkojen välinen kommunikointi tapahtuu.

3.4 Näkyvyysalue

Näkyvyysaluetta tarkasteltiin jo hieman aikaisemmin header guard -mekanismin yhteydessä. Oletuksena käytettävä näkyvyysalue on globaali näkyvyysalue, jossa määritetyt esineet esimerkiksi muuttujat, funktiot tai luokat ovat kaikkien lähdekoodien käytettävissä. Globaalia näkyvyysalueen käyttöä kuitenkin pyritään välttämään, jotta määritykset pysyvät paremmin luokiteltuna ja välttääkseen niiden ylikirjoittamista vääristä paikoista. Tietyt SystemVerilogin elementit luovat uuden näkyvyysalueen, jotka ovat standardissa IEEE 1800-2017 (2018) määritettynä olevan

- *module*-avainsanalla luodut moduulit
- *interface*-avainsanalla luodut väylät
- *program*-avainsanalla luodut ohjelmalohkot
- *checker*-avainsanalla luodut tarkistuslohkot
- *package*-avainsanalla luodut paketit
- *class*-avainsanalla luodut luokat
- *task*-avainsanalla luodut tehtävät
- *function*-avainsanalla luodut funktiot
- *begin-end*-lohkot
- *fork*-lohkot
- *generate*-lohkot.

Tunniste eli nimi tulee olla vain kerran määritettynä yhdellä näkyvyysalueella. Tämä tarkoittaa, että esimerkiksi useammalla muuttujalla funktion sisällä ei voi olla samaa nimeä eli tunnistetta tai luokan sisällä määritetyn funktion nimi ei voi olla sama kuin luokassa määritetyn muuttujan tai toisen funktion nimi. Tunnisteita voidaan myös kutsua käyttämällä tunnisteen hierarkkista polkua, jossa polun eri tunnisteet erotellaan pisteellä.

Tunnisteiden hierarkkisen polun lisäksi on olemassa luokille luokan näkyvyysalueen resoluutio operaattori *::* eli kaksi kaksoispistettä. Sitä käytetään tarkentamaan määritystä luokan näkyvyysalueella ja myös usein määrittääkseen metodeja luokan ulkopuolelle. Toinen syy luokan näkyvyysalueen resoluutio operaattorin käytölle on, että useammalla luokalla voi olla samannimisiä muuttujia tai metodeja. Erottaakseen, minkä luokan muuttujaa tai metodia ollaan kutsumassa,

voidaan käyttää luokan näkyvyysalueen resoluutio operaattoria. (IEEE 1800-2017 2018, 720–730.)

Hyvänä ohjesääntönä on kirjoittaa luokkien metodit niiden ulkopuolelle, jotta lähdekoodista tulisi luettavampaa. Metodit saattavat olla hyvinkin pitkiä ja niiden löytäminen luokan sisältä on työlästä varsinkin, jos lähdekoodien lukija ei tiedä entuudestaan olemassa olevia metodeja. SystemVerilogiin ja metodologiaan erikoistunut verifiointi-insinööri Chris Spear selittää kirjassaan, kuinka metodit luodaan luokan ulkopuolelle. Luokan metodit voidaan esitellä luokassa eli voidaan luoda metodien prototyypit *extern*-avainsanalla ja siirtää koko metodi sekä sen funktionaalisuus luokan ulkopuolelle käyttämällä luokan näkyvyysalueen resoluutio operaattoria (Spear 2008, 139–140).

Avainsana *extern* indikoi, että luokan metodi tulee löytymään luokan ulkopuolelta. Luokan ulkopuolelle määritetyn metodin täytyy kuitenkin vastata luokassa määriteltyä prototyyppiä täsmällisesti tietyin poikkeuksin. Metodien prototyypeille voidaan myös lisätä tarkenne kuten *virtual, protected tai local*. (IEEE 1800-2017 2018, 187–190.)

Yksi hyödyllinen avainsana viittaamaan hierarkkisesti SystemVerilogissa on *this*, jota ei tulisi kuitenkaan käyttää liikaa. Spear avaa avainsanan tarkoitusta kirjassaan kysymällä, jos ollaan luokassa ja halutaan viitata luokkatasolla olevaan olioon, kuinka tämä onnistuisi? Esimerkiksi luokan metodissa halutaan asettaa samanniminen jäsenmuuttuja kuin metodin argumentti. Tällöin avainsanalla *this* voidaan viitata jäsenmuuttujaan ja asettaa sen arvoksi samanniminen argumenttimuuttuja eli pystytään erottamaan metodin ja luokan muuttuja toisistaan (kuva 9). (Spear 2008, 143–144.)

```
1   `include "luokka_1.svh"
2   module lohko;
3     int x;
4     luokka_1 olio;
5
6     initial begin
7       olio = new();
8       x = 1;
9       olio.tulosta(x);
10    end
11  endmodule
12

1   class luokka_1;
2     int x = 0;
3
4     function void tulosta(int x);
5       $display("Argumentti x:%0d, Luokka x:%0d",x, this.x);
6     endfunction
7   endclass
8
# Argumentti x:1, Luokka x:0
```

KUVA 9. Esimerkki avainsanan *this* käyttämisestä

3.5 Lohkojen ilmentymien luominen

SystemVerilogilla luodaan lohkohierarkia, kun yhden lohkon sisään luodaan toisen rakennelohkon ilmentymä, jolloin luodaan uusi lohkohierarkiataso. Sisäkkäisten ja rinnakkaisten lohkojen kommunikointi tapahtuu liittämällä niiden portit toisiinsa signaaleilla. (IEEE 1800-2017 2018, 51–52.)

Korkeimmalla hierarkiassa oleva lohko eli top-tason lohko on lohko, jota ei sisällytetä toisiin lohkoihin. Esimerkiksi järjestelmäpiirin top-tason lohko on itse järjestelmäpiiri ja sen sisälle luodaan sen osalohkojen ilmentymiä. On myös mahdollista olla useampia top-tason lohkoja.
Testipenkkien toteutuksessa on yksi tietty testipenkin osa, jossa luodaan lohko toisen lohkon sisään ja se on DUT wrapper eli lohko, joka sisältää itse DUT:n. DUT wrapper -lohkoa käytetään, jotta DUT:n luominen testipenkkiin olisi yksinkertaisempaa ja pitääkseen sen ilmentymän luominen eriteltynä testipenkkilohkosta. DUT wrapper -lohkoon siis luodaan verifioitavan lohkon ilmentymä, jonka portteihin kiinnitetään testipenkin väylien signaaleja sekä tehdään muut tarvittavat toimenpiteet toimivan DUT:n luomiseksi. Riippuen käytettävästä simulaattorista, myös VHDL-kuvauksien ilmentymiä voidaan luoda DUT wrapper -lohkoon. Tätä kutsutaan monikieliseksi simuloinniksi.

3.6 Käännösjärjestys

Kaikki lähdekoodit ovat oletuksena erillisiä kokonaisuuksia. Yksi lähdekooditiedosto ei voi viitata toiseen lähdekooditiedostoon sisällyttämättä ensin `*include*-kääntäjädirektiivillä toisen lähdekoodeja, joihin se viittaa. Esimerkiksi moduuli ei voi luoda luokasta oliota tai viitata jäsenmuuttujiin, ellei luokkaa ole tehty moduulille näkyväksi.

Lähdekoodien näkyvyydet ja riippuvuudet toisiinsa aiheuttavatkin usein käännöksissä virheitä. Lähdekoodit on käännettävä oikeassa järjestyksessä ja myös tästä syystä suositaan pakettien käyttöä lähdekoodien jakamiseksi. Paketeissa on myös kutsuttava `*include*-kääntäjädirektiivit oikeassa järjestyksessä, mutta järjestyksen muuttaminen ja pakettien ylläpito on huomattavasti helpompaa.

4 UVM

Tässä luvussa esitellään UVM:ää ja sen kantaluokkakirjastoa. Luvussa pyritään esittelemään kaikki välttämätön UVM:n sisältö, jotta lukija ymmärtäisi toteutuksessa esitettyjä asioita.

4.1 Päämäärä

Useat tavat luoda verifiointiympäristöjä tekevät niistä virheille alttiita ja vievät aikaa toteuttaa. UVM:n tarkoituksena on luoda standardoitu SystemVerilogiin pohjautuva yhteistoimiva verifiointimenetelmä, jolla toteuttaa testipenkkejä piirien verifioimiseksi. Standardi vakiinnuttaa UVM:n ohjelmointirajapinnat, jotka määrittelevät kantaluokkakirjaston testipenkkien toteuttamiseksi. (IEEE 1800.2 2020, 12.) UVM on IEEE-standardi, jonka viimeisin revisioitu versio IEEE 1800.2-2020 julkaistiin vuonna 2020.

4.2 UVM-hierarkia

UVM:n hierarkiaan kuuluu neljä pääkantaluokkaa, josta muut luokat periytyvät: *uvm_void, uvm_object, uvm_transaction* ja *uvm_component*. Luokat *uvm_component* ja *uvm_transaction* periytyvät *uvm_object*-luokasta, joka periytyy itse korkeimmalla hierarkiassa olevasta *uvm_void*-luokasta. Luokasta *uvm_component* periytyviä luokkia kutsutaan UVM-komponenteiksi ja luokasta *uvm_object* kutsutaan UVM-objekteiksi.

Luokka *uvm_void* on abstrakti luokka kaikille UVM-luokille, eikä sisällä jäsenmuuttujia tai metodeja. Luokat, jotka periytyvät suoraan *uvm_void*-luokasta, eivät peri mitään UVM:n kantaluokkakirjaston ominaisuuksia. Kantaluokkakirjaston ominaisuudet periytyvät *uvm_object*-luokasta.

Luokka *uvm_object* toimii kantaluokkana kaikille hierarkkisille luokille UVM:ssä ja määrittää tietyt yleismetodit, jotka periytyvät suoraan tai epäsuorasti aliluokilleen.

Luokka sisältää esimerkiksi metodit *create, copy* ja *compare*. Luokan rakentajafunktio *new* ottaa *string*-tyyppisen *name*-muuttujan argumentikseen. Muuttuja toimii tunnisteena luotavalle oliolle ja voidaan nimetä uusiksi tarvittaessa *set_name*-funktiolla.

Luokka *uvm_component* perii kaikki metodit hierarkiassa olevalta ylemmältä luokaltaan eli *uvm_report_object*-luokalta, joka periytyy *uvm_object*-luokasta. Periytyvien metodien lisäksi, *uvm_component* tarjoaa myös rajapinnat

- hierarkialle
- vaiheistukselle
- hierarkkiselle raportoinnille
- transaktioiden tallennukselle
- tehtaan käytölle.

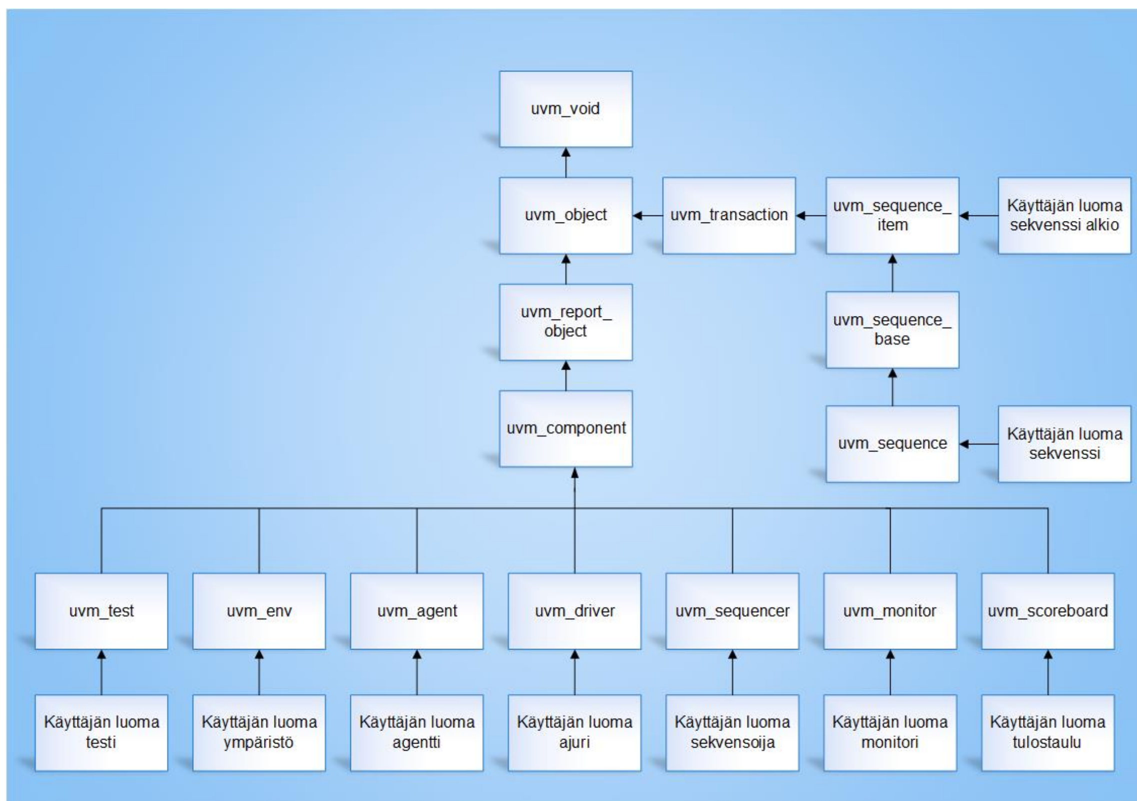
UVM:n kantaluokkakirjasto tarjoaa valmiiksi määriteltyjä komponenttityyppejä, jotka periytyvät *uvm_component*-luokasta. Komponentit kestävät koko simulaation ajan toisin kuin transaktiot. Valmiiksi määritetyt komponenttityypit ovat

- *uvm_test*
- *uvm_env*
- *uvm_agent*
- *uvm_monitor*
- *uvm_scoreboard*
- *uvm_driver*
- *uvm_sequencer*
- *uvm_push_driver*
- *uvm_subscriber*.

UVM-komponenttien rakentajafunktioille tulee määrittää argumenteiksi *string name* ja *uvm_component parent*. Argumentiksi annettava muuttuja *name* on sama kuin *uvm_object*-luokan yhteydessä ja *parent* on komponentin luovan luokan nimi. Kaikkien luokkien, jotka periytyvät *uvm_component*-luokasta, on kutsuttava *super.new(name,parent)* rakentajafunktiossaan.

Suoraan *uvm_object*-luokasta periytyvä *uvm_transaction*-luokka puolestaan on kantaluokkana kaikille transaktioille eli tiedonsiirtotapahtumille eri komponenttien välillä. Periytyvien *uvm_object*-luokan metodien lisäksi, *uvm_transaction* lisää ajoituksen ja tallentamisen rajapinnan. (IEEE 1800.2-2020 2020, 18–19, 27, 159–160, 197.)

Testipenkin toteutusta varten tärkeimmät luokat ovat *uvm_component*-luokan valmiiksi määritetyt komponenttityypit sekä *uvm_transaction*-luokan *uvm_sequence*- ja *uvm_sequence_item*-luokat. Testipenkin toteutuksessa laajennetaan edellä mainittuja luokkia eli käyttäjän luomat luokat periytyvät näistä kantaluokista ja lopputuloksena on kuvion 3 mukainen yksinkertaistettu UVM-luokkahierarkia.



KUVIO 3. Yksinkertaistettu esitys UVM-luokkahierarkiasta (UVM Introduction n.d., muokattu)

4.3 UVM-tietokannat

Resurssi on UVM:ssä parametrisoitu säiliö, joka sisältää mielivaltaista dataa esimerkiksi konfiguroimaan komponentteja tai syöttämään dataa sekvensseille. Tallennettavat resurssisäiliöt parametrisoidaan halutulla resurssin tietotyypillä, joka voi olla mikä tahansa sallittu SystemVerilogin tyyppi kuten skalaari, olion kahva, jono, lista tai jopa virtuaalinen väylä. Resursseja voidaan hakea nimellä tai tyypillä ja niiden näkyvyyttä voidaan rajata vain halutuille UVM-testipenkin luokille.

UVM sisältää kaksi luokkaa, jotka luovat rajapinnan yksinkertaistamaan resurssien käyttämistä ja mahdollistaa niiden jakamisen eri komponenttien välillä. Nämä luokat ovat *uvm_resource_db* ja siitä periytyvä *uvm_config_db*-luokka, joita kutsutaan UVM:n resurssi- ja konfiguraatiotietokannoiksi. Luokka *uvm_resource_db* yhdistää seuraavien resurssiluokkien toiminnallisuudet ja toimii niille rajapintana

- *uvm_resource_types*
- *uvm_resource_base*
- *uvm_resource_pool*
- *uvm_resource#(T)*.

Tietokanta *uvm_resource_db* tarjoaa staattisia metodeja kuten *set*. Funktio *set* luo uuden resurssin ja sisältää argumentit

- *input string scope* eli resurssin näkyvyysalue
- *input string name* eli resurssin nimi
- *T val* eli resurssin arvo, jossa T on parametrina annettu tyyppi
- *input uvm_object accessor = null* asetetaan nykyiseksi näkyvyysalueeksi.

Resurssien arvojen lukemiseksi on olemassa funktiot *read_by_name* ja *read_by-_type*. Funktiolla *read_by_name* haetaan tietotyyppiä nimellä ja *read_by_type*-funktiolla haetaan tyypillä. Funktiolle *read_by_name* annetaan argumenteiksi näkyvyysalue, resurssin nimi sekä muuttuja, johon tallentaa luettu resurssi. Funktio *read_by_type* ottaa argumentikseen vain resurssin näkyvyysalueen ja muuttujan. Metodeja kutsuessa on käytettävä näkyvyysalueen resoluutio operaattoria, koska metodit ovat staattisia. Kuvassa 10 esitetään esimerkki *set*- ja *read_by_type*-funktioista.

```
1   import uvm_pkg::*;
2   import my_test_pkg::*;
3   `include "uvm_macros.svh"
4
5   module lohko;
6     `include "luokka_1.svh"
7
8     int x = 17;
9     luokka_1 olio;
10
11    initial begin
12      // Aseta arvo 'x' näkyvyysalueelle nimeltä 'kokonaisluvut'
13      uvm_resource_db#(int)::set("kokonaisluvut","x",x);
14      olio = new();
15      olio.tulosta();
16    end
17  endmodule
18
1   class luokka_1;
2     int x;
3
4     function void tulosta();
5       uvm_resource_db#(int)::read_by_type("kokonaisluvut",x);
6       $display("Tietokannasta haettu arvo: %0d",x);
7     endfunction
8   endclass
9
#
# Tietokannasta haettu arvo: 17
```

KUVA 10. Resurssin asettaminen ja hakeminen *uvm_resource_db*-tietokannasta

Tietokanta *uvm_config_db* periytyy *uvm_resource_db*-luokasta ja sen tarkoituksena on toimia rajapintana UVM-komponenttien konfiguroimiseksi. Tietokanta sisältää omat metodit ilmentymien tallentamiseksi ja hakemiseksi. Sen funktio *set* tallentaa tai päivittää jo olemassa olevan ilmentymän ja ottaa parametreikseen

- *uvm_component cntxt*
- *string inst_name*
- *string field_name*
- *T value*.

Tallentaessa tai päivittäessä ilmentymää, funktion argumentit *cntxt* ja *inst_name* muodostavat yhdessä ilmentymän näkyvyysalueen *cntxt.inst_name*. Ensimmäinen argumentti *cntxt* määrittää, mistä ilmentymää lähdetään etsimään tietokannasta ja *inst_name* määrittää näkyvyysalueen. Argumentti *field_name* on tietokenttä, jota halutaan muuttaa ja *value* on sen arvo.

Tietoja puolestaan haetaan tietokannasta *get*-funktiolla, joka palauttaa arvon yksi, jos tietokannasta hakeminen onnistui tai nollan, jos ei onnistunut. Funktiolle annetaan argumenteiksi

- *uvm_component cntxt*
- *string inst_name*
- *string field_name*
- *inout T value*.

Muuttujalle *value* annetaan muuttujan nimi, johon haettu data tietokannasta halutaan säilyttää. (IEEE 1800.2-2020 2020, 392–406; uvm_config_db examples n.d.) Tietokannasta voidaan esimerkiksi hakea *string* tyyppinen resurssi muuttujaan nimeltä *etunimi* kuvan 11 esittämällä käskyllä. Käsky etsii *uvm_component*-tasolta *nimet*-säiliöstä nimeä *etunimi_1* ja tallentaa sen arvon *etunimi* muuttujaan.

```
1   import uvm_pkg::*;
2   import my_test_pkg::*;
3   `include "uvm_macros.svh"
4
5   module lohko;
6     `include "luokka_1.svh"
7
8     luokka_1 olio;
9
10    initial begin
11      // Aseta arvo 'Pekka Pouta' näkyvyysalueelle 'null' eli kaikille komponenteille säiliöön 'nimet' nimellä 'etunimi_1'
12      uvm_config_db#(string)::set(null,"nimet","etunimi_1","Pekka Pouta");
13      olio = new();
14      olio.tulosta();
15    end
16  endmodule
17
1   class luokka_1;
2     string etunimi;
3     function void tulosta();
4       // Kutsu palauttaa 1 jos onnistui ja 0 jos ei
5       // Lopetetaan simulaatio, jos haku ei onnistunut
6       if (!uvm_config_db#(string)::get(null, "nimet", "etunimi_1",etunimi)) begin
7         `uvm_fatal("luokka_1","Haku tietokannasta ei onnistunut")
8       end
9       $display("Tietokannasta haettiin: %0s",etunimi);
10    endfunction
11  endclass
12
```

```
#
# Tietokannasta haettiin: Pekka Pouta
```

KUVA 11. *uvm_config_db*-tietokannan käyttäminen

Toisena esimerkkinä voidaan luoda virtuaalisen väylän ilmentymä ja liittää se väylän ilmentymään kuvan 12 mukaisella käskyllä. Kuvassa *vayla* on väylän nimi, *virtuaali_vayla* on luotavan virtuaalisen väylän ilmentymän nimi ja *vayla_1* on erään väylästä *vayla* luodun ilmentymän nimi.

```
1   import uvm_pkg::*;
2   import my_test_pkg::*;
3   `include "uvm_macros.svh"
4
5   module lohko;
6     `include "luokka_1.svh"
7     `include "vayla.sv"
8
9     vayla vayla_1();
10    luokka_1 olio;
11
12    initial begin
13      // Aseta väylän ilmentymä tietokantaan
14      uvm_config_db#(virtual vayla)::set(null, "*", "virtuaali_vayla", vayla_1);
15      olio = new();
16      olio.tulosta();
17      olio.virtuaali_vayla.signaali = 1;
18      $display("vaylan arvo on: %0d", olio.virtuaali_vayla.signaali);
19    end
20  endmodule
21
```

```
1   class luokka_1;
2
3     virtual vayla virtuaali_vayla;
4
5     function void tulosta();
6       // Kutsu palauttaa 1 jos onnistui ja 0 jos ei
7       // Lopetetaan simulaatio, jos haku ei onnistunut
8       if (!uvm_config_db#(virtual vayla)::get(null, "*", "virtuaali_vayla",virtuaali_vayla)) begin
9         `uvm_fatal("luokka_1","Haku tietokannasta ei onnistunut")
10      end
11      $display("Tietokannasta haettiin: %0d",virtuaali_vayla.signaali);
12    endfunction
13  endclass
14
```

```
# Tietokannasta haettiin: x
# vaylan arvo on: 1
```

KUVA 12. Väylien liittäminen moduulien ja luokkien välille UVM:ssä

Tällöin väylään *vayla* osoitetaan *get*-funktiolla haetulla virtuaalisella väylällä. Funktion palautusarvoa voidaan hyödyntää asettamalla funktion ympärille ehtolause, joka tarkistaa tietokannan haun tuloksen. Ehtolauseella voidaan lopettaa simulaatio *`uvm_fatal*-raportointimakrolla, jos väylän arvoa ei saada jostain syystä. (UVM Config db n.d.) Tällä varmistetaan, että väylän yhteys toimii simulaation aikana ja komponentti voi lukea tai kirjoittaa väylän signaaleita.

Muita hyviä käyttötarkoituksia tietokannoille on olioiden kahvojen jakaminen eri komponenttien välillä. Esimerkiksi ympäristöluokassa saattaa olla toisesta luokasta luotu olio, joka halutaan jakaa käytettäväksi testiluokassa.

4.4 UVM-tehdas

UVM-tehdas on luokka, jota käytetään luomaan *typedef* ja makrokutsuilla tehtaalle rekisteröityjä UVM-objekteja ja -komponentteja. Sen mekanismi parantaa testipenkin joustavuutta mahdollistamalla testipenkissä luotujen olioiden tyypin korvaamisen olioilla, joiden luokat periytyvät jo olemassa olevan olion luokasta.

Yleensä oliot luodaan SystemVerilogissa *new*-funktiolla. Ongelmana on, että käsky on staattinen ja jokainen funktiokutsu olisi muutettava erikseen, jos kyseisen olion tyyppi haluttaisiin muuttaa. Tehtaassa voidaan korvata rekisteröityjä olioita käyttämällä tehtaan makroja, jotka muuttavat rekisteröidyn olion tyypin.

Tehtaan käyttämiseksi tehdään kolme toimenpidettä. Ensin UVM-objektit ja -komponentit rekisteröidään tehtaalle, toiseksi ne luodaan UVM-komponenttiluokkiin käyttämällä tehtaan metodeja ja viimeiseksi konfiguroidaan tarvittaessa tehtaalle rekisteröityjä tietotyyppejä korvaamalla niiden tyyppejä ja ilmentymiä.

Jokaisessa käyttäjän luomassa luokassa, joka periytyy *uvm_object*-luokasta, tulisi käyttää *utils*-makroja rekisteröimään tehtaalle UVM-objekteja ja -komponentteja. Makro *`uvm_object_utils* voidaan käyttää rekisteröimään UVM-objekteja ja UVM-komponentteja varten makroa *`uvm_component_utils*. Näiden kahden makron lisäksi on myös muita makroja, joilla rekisteröidä objekteja ja komponentteja. Kummallekin esitellylle makrolle annetaan argumentiksi luokan nimi, joka halutaan rekisteröidä tehtaalle ja, josta halutaan luoda olioita tehtaan avulla.

UVM-komponenttien sisään luodaan toisia komponentteja tai UVM-objekteja *create*-metodilla. Metodi ottaa argumentikseen *string name* eli luotavan olion nimi luodessa UVM-objekteja, mutta UVM-komponentteja luodessa, metodille annetaan lisäksi argumentti *uvm_component parent* eli käskyä kutsuvan komponentin nimi. UVM-komponenttien lähdekoodeissa luodaan ensin muuttuja luotavan UVM-komponentin olion kahvaa varten, jonka arvoksi asetetaan metodin *create* kutsu. Kuvassa 13 on esimerkki uuden komponentin luomisesta toiseen komponenttiin.

```
import uvm_pkg::*;
import my_test_pkg::*;
`include "uvm_macros.svh"

module lohko;
  `include "luokka_2.svh"
  `include "luokka_1.svh"
  initial begin
    run_test("testi");
  end
endmodule
```

```
class ymparisto extends uvm_env;
  `uvm_component_utils(ymparisto)
  extern function new(string name="ymparisto", uvm_component parent);
  extern function void tulosta();
endclass

// Luo uuden ymparisto olion
function ymparisto::new(string name="ymparisto", uvm_component parent);
  super.new(name,parent);
endfunction : new

// tulostaa 'Luokassa ymparisto'
function void ymparisto::tulosta();
  `uvm_info("ymparisto", "Luokassa ymparisto", UVM_LOW);
endfunction : tulosta
```

```
class testi extends uvm_test;
  `uvm_component_utils(testi)                                         // Rekisteröi luokka UVM-tehtaalle
  ymparisto testipenkki_ymparisto;                                    // Muuttuja sisältämään olion kahvan

  extern function new(string name = "testi", uvm_component parent = null);  // Rakentajafunktion prototyyppi
endclass

// Luo uuden testi olion
function testi::new(string name = "testi", uvm_component parent = null);
  super.new(name, parent);                                            // kutsu kantaluokan new-metodia
  `uvm_info("testi", "Testiluokassa", UVM_LOW);                       // Käytä UVM raportointimekanismia
  testipenkki_ymparisto = ymparisto::type_id::create("testipenkki_ymparisto", this); // Luodaan olio käyttämällä tehdasta
  testipenkki_ymparisto.tulosta();                                    // kutsu metodia tulosta
endfunction
```

```
# UVM_INFO \luokka_1.svh(11) @ 0: uvm_test_top [testi] Testiluokassa
# UVM_INFO \luokka_2.svh(14) @ 0: uvm_test_top.testipenkki_ymparisto [ymparisto] Luokassa ymparisto
# UVM_INFO @ 0: reporter [RNTST] Running test testi...
#
#
#
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    5
# UVM_WARNING :    2
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [RNTST]     1
# [UVM/COMP/NAME]     2
# [UVM/COMP/NAMECHECK]     1
# [UVM/RELNOTES]     1
# [testi]     1
# [ymparisto]     1
#
# ** Note: $finish    : /uvm_root.svh(578)
#    Time: 0 ps  Iteration: 259  Instance: /lohko
```

KUVA 13. UVM-komponentin luominen toiseen komponenttiin

Lopuksi *uvm_test*-luokasta periytyvästä käyttäjän luomasta testiluokasta voidaan korvata tehtaan kautta olioita toisilla eli konfiguroida käytettävät UVM-komponentit ja -objektit ennen niiden luomista. Olioiden korvaamiseksi on olemassa metodit kuten *set_inst_override_by_type*. (IEEE 1800.2-2020 2020, 69–81,369–372; UVM Factory Override n.d.; UVM Factory n.d.)

4.5 UVM-vaiheet

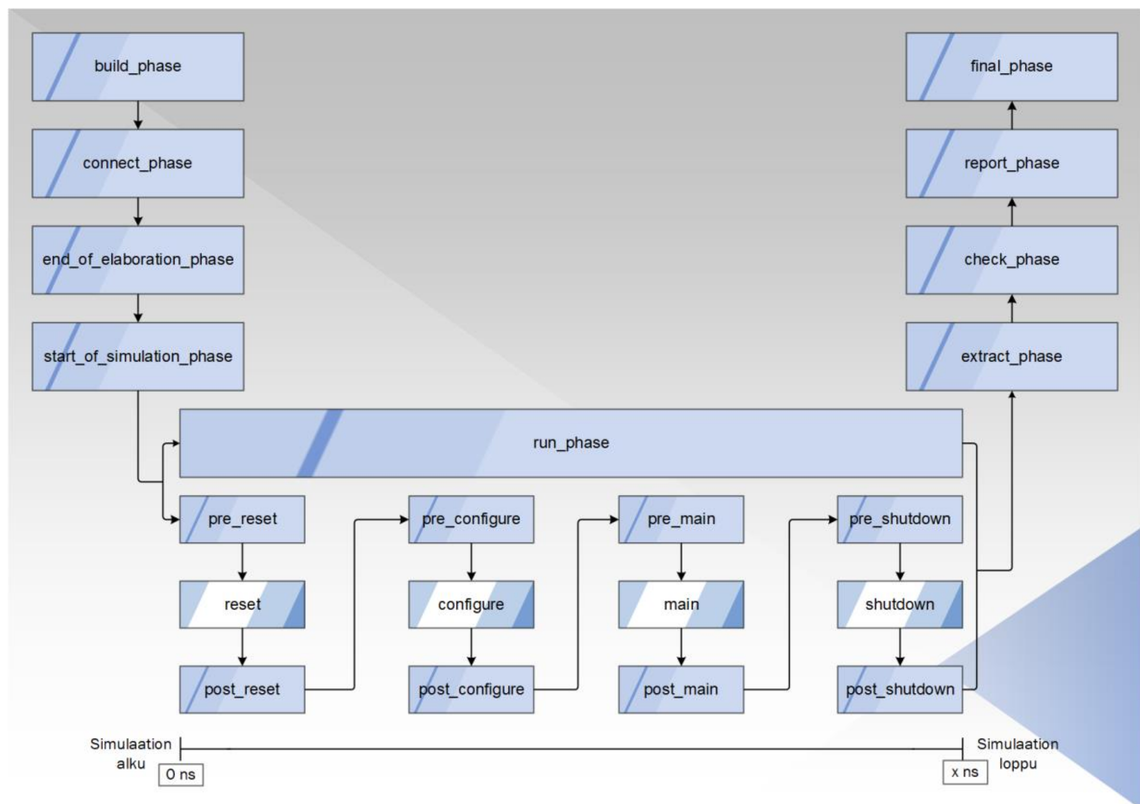
UVM tarjoaa valmiiksi määriteltyjä vaiheita simulaatioiden ajon ajaksi. UVM-vaiheet ovat UVM-komponenttien perimiä metodeja, joilla jaksotetaan simuloinnin kulkua ja synkronoidaan jokaisen komponentin toimenpiteet. UVM:n yleisvaiheet voidaan ryhmittää koontiaikaiseksi, ajonaikaiseksi tai poiskorjausaikaiseksi vaiheeksi. Yleisvaiheet ja niiden tarkoitukset ovat taulukoituna taulukossa 4.

TAULUKKO 4. UVM:n yleisvaiheet (UVM Phases n.d, muokattu)

Vaihe	Tarkoitus
build_phase	Komponentit kootaan ja konfiguroidaan
connect_phase	Komponenttien TLM portit liitetään toisiinsa
end_of_elaboration_phase	Voidaan esittää testipenkin ympäristön topologia tai avata tiedostoja
start_of_simulation_phase	Tehdään viimeiset valmistelut ennen simulaation ajoa kuten muuttujien alustamiset
run_phase	Simulaation ajonaikainen tehtävä, jossa suoritetaan itse DUT:n stimulointi
extract_phase	Poiskorjausvaihe, jossa korjataan pois jäänyt data ja informaatio tulostaululta ja lasketaan testin tuloksen tilastot
check_phase	Tarkistetaan, ettei ylimääräistä dataa ole jäänyt ja testi on hyväksytty tai hylätty
report_phase	Raportoidaan testin tulokset ja kirjoitetaan ne lokitiedostoon
final_phase	Suljetaan kaikki tiedostot ja valmistellaan simulaattorista poistuminen

UVM-vaiheilla on tietty suoritusjärjestys alkaen komponenttien kokoamisesta build_phase-vaiheessa ja loppuen final_phase-vaiheeseen. UVM-vaiheiden suoritusjärjestyksellä varmistutaan, että kaikki komponentit ovat suorittaneet samanlaiset toimenpiteet samanaikaisesti eli komponentit synkronoidaan.

Yleisvaiheiden lisäksi, vaihe run_phase sisältää myös sisäisiä vaiheita, jotka suorittavat rinnakkaisesti run_phase-vaiheen kanssa ja voidaan käyttää tarvittaessa. Vaihe run_phase sisältää esimerkiksi vaiheen uudelleenkäynnistämään DUT sekä toisen vaiheen konfiguroimiselle. UVM-vaiheiden suoritusjärjestys on esitetty kuviossa 4.



KUVIO 4. UVM-vaiheet ja niiden suoritusjärjestys (UVM Phases n.d., muokattu)

Vaihe *run_phase* on myös ainoa vaihe, joka on tyypiltään tehtävä ja voi kuluttaa simulointiaikaa. Tämän takia jokaisen komponentin *run_phase*-vaiheen alussa, jonka vaihetta halutaan pitää yllä, täytyy kutsua metodia *phase.raise_objection*. Metodin *phase.raise_objection* käyttö tiedottaa muille komponenteille, että komponentti ei ole suorittanut vielä kaikkea. Samalla tavalla, kun komponentti on suorittanut *run_phase*-vaiheessa tarvittavat asiat, sen on kutsuttava metodia *phase.drop_objection*.

Metodeille annetaan argumentiksi UVM-komponentin luokka, jonka *run_phase*-vaihetta on ylläpidettävä. Tämä voidaan tehdä yksinkertaisesti antamalla argumentiksi avainsana *this*. Metodien *phase.raise_objection(this)* ja *phase.drop_objection(this)* kanssa on oltava tarkkana, ettei *run_phase*-vaihe jää ikuiseen silmukkaan. (IEEE 1800.2-2020 2020, 162–165; UVM Phases n.d.; UVM Common Phases n.d.; UVM Run-Time Phases n.d.)

4.6 TLM-rajapinta

UVM sisältää SystemC kaltaisen TLM (transaction-level modeling) -rajapinnan eri kokonaisuuksien väliseen kommunikointiin. TLM-rajapinta on jaettu kahteen osaan, TLM 1 ja TLM 2, joiden käyttöä rajapinta yksinkertaistaa. TLM 1:stä käytetään siirtämään TLM-porttityyppien kautta mielivaltaisista tietotyypeistä koostuvia viestejä ja TLM 2:hta siirtämään standardoituja transaktio-olioita, joiden nimi on *payload* ja mallintavat protokollia.

TLM-liitännät voivat olla *blocking*, *non-blocking* tai niiden yhdistelmä (eng. *combination)* ja määrittelevät kommunikointitavan kokonaisuuksien välillä. Eri liitäntätyypit ja niiden erot selitetään taulukossa 5.

TAULUKKO 5. TLM-liitäntöjen kommunikointitavat

liitäntätyyppi	selite
blocking	Metodit ovat tehtäviä, koska saattavat kuluttaa aikaa, eivätkä palaa eli ovat estyneitä, kunnes transaktio on onnistunut
non-blocking	Metodit ovat funktiota, eivätkä kuluta aikaa sekä palaavat heti. Transaktio saattaa epäonnistua, jolloin metodi palauttaa epäonnistumisesta statuksen.
combination	On *blocking* ja *non-blocking* kommunikointitapojen yhdistelmä

TLM-rajapinnalla voidaan luoda yksi- tai kaksisuuntaisia *port-, export-* tai *implementation*-porttityyppejä, joiden liitäntätyyppi voi olla jokin edellä mainituista tyypeistä. Eri porttityypit ja niiden käyttötarkoituksia esitellään taulukossa 6.

TAULUKKO 6. Porttityypit ja niiden käyttötarkoitukset

porttityyppi	käyttötarkoitus
port	Käytetään komponenteilla, joiden tarvitsee tai käyttävät liitäntää käynnistämään transaktioita
export	Käytetään komponenteilla, jotka välittävät transaktioita
imp	Käytetään komponenteilla, jotka tarjoavat metodit määritellylle liitännälle

Uudet TLM-porttityypit muodostetaan luomalla ensin luokan *uvm_*_port, uvm_*_export* tai *uvm_*_imp* tyyppiä oleva muuttuja. Merkillä * tarkoitetaan TLM-porttityypin rajapintaluokkaa, jota halutaan käyttää ja, josta voidaan luoda olioita. Porttityyppien rajapintoja ovat

- *put*
- *get*
- *peek*
- *get_peek*
- *analysis*
- *transport*
- *master*
- *slave.*

Näistä rajapinnoista*, transport, master* ja *slave* ovat kaksisuuntaisia ja muut ovat yksisuuntaisia. Jokaiselle porttityypille on olemassa siis luokat rajapinnoille sekä niiden jokaista mahdollista liitäntätyyppiä varten. Esimerkiksi *port*-porttityypille on olemassa *put*-rajapinnasta *uvm_blocking_put_port-, uvm_nonblocking_put_port-* sekä *uvm_put_port*-luokat eri liitäntätyyppejä varten ja *export*-porttityypille on olemassa *peek*-rajapinnasta luokat *uvm_blocking_peek_export-, uvm_non-blocking_peek_export-* sekä *uvm_peek_export*-luokka. Poikkeuksena on *analysis*-rajapinta, jolle ei ole olemassa *non-blocking* tai *blocking* liitäntätyyppiä. Luokat määrittävät porttityypin, rajapinnan sekä sen liitäntätyypin.

Uuden *port*- tai *export*-porttityypin olion luomiseksi käytetään valitun rajapinnan luokan rakentajafunktiota, joka sisältää neljä argumenttia. Argumentit ovat *string name* ja *uvm_component parent*, jotka ovat porttityypin luokan nimi ja porttityypin luovan luokan nimi sekä *int min_size=1* ja *int max_size=1*, jotka ovat minimi- ja maksimimäärä porttityyppejä, joita voi olla liitettynä luotavaan porttiin. Porttien minimi- ja maksimimäärää voi vaihtaa antamalla argumentiksi uuden arvon, mutta oletuksena voi olla vähintään ja enintään yksi portti liitettynä.

Porttityypin *imp* rakentajafunktio *new* ottaa argumenteikseen *string name* ja *IMP parent*, jossa *name* on porttityypin luokan nimi ja *IMP* on porttityypin luovan luokan nimi. Porttityyppi *imp* on siitä erikoinen, että sille on määritettävä *write*-metodi siinä luokassa, jossa portti on luotu. Metodi *write* kutsutaan aina, kun uusi olio on saatu *imp*-porttityypillä vastaan ja siinä määritellään, mitä olion datalla tulisi tehdä.

TLM-rajapinnan luokka *uvm_tlm_if_base* määrittää kaikki sen metodit ja sisältää esimerkiksi metodin *write*, joka käynnistää transaktion päätepisteessä *imp*-porttityypin sisältävän luokan metodin *write*. Metodin *write* kutsulla lähetetään transaktio *port*-porttityypin kautta kaikille porttia kuunteleville porttityypeille. Metodi *write* on funktio, joten sen tulee suorittaa lähetys ilman esteitä ja sille annetaan argumentiksi transaktion tietotyyppi, joka halutaan lähettää.

UVM:n TLM-rajapinnalla voidaan myös luoda FIFO (first in, first out) -puskureita *uvm_tlm_fifo_base*-kantaluokalla, joka toimii rajapintana kaikille FIFO-luokille ja tarjoaa niiden yleismetodit. Kantaluokasta *uvm_tlm_fifo_base* periytyy luokat *uvm_tlm_fifo* ja *uvm_tlm_analysis_fifo.*

Erilaiset TLM-porttityypit ja FIFOt liitetään toisiinsa käyttämällä niiden TLM-rajapinnan *connect*-funktiota. (IEEE 1800.2-2020 2020, 7, 33, 120–159.) Esimerkiksi *A.transaktio_out.connect(B.transaktio_in)* käskyllä liitettäisiin *A* ja *B* UVM-komponenttien portit *transaktio_out* ja *transaktio_in*.

4.7 Testipenkkilohko

Testipenkkilohko ei kuulu Accelleran julkaisemaan UVM:n kantaluokkakirjastoon, mutta on välttämätön käyttääkseen itse UVM:ää. Testipenkkilohko on käyttäjän oma luoma testipenkin ylin lohko ja sen tärkeimpänä tehtävänä on käynnistää UVM-testi kutsumalla UVM:n funktiota *run_test*.

Funktiokutsu luo koko testipenkin, simuloi DUT:n toimintaa ja tuottaa lopuksi testin tulokset. Funktiolle *run_test* annetaan ajettavan testin nimi lainausmerkeissä tai se voidaan antaa simulaattorikomentojen UVM_TEST-kytkimellä. Testipenkkilohkossa testin käynnistämisen lisäksi luodaan *uvm_config_db*-tietokantaan testipenkin UVM-komponenteissa käytettävät virtuaaliset väylät, jotka liitetään DUT:n väyliin.

4.8 DUT wrapper

DUT wrapper ei myöskään kuulu Accelleran julkaisemiin lähdekoodeihin. DUT wrapper on lohko, jonka sisälle luodaan ilmentymä verifioitavasta lohkosta eli DUT:sta. DUT wrapper -lohkon argumenteiksi asetetaan tarvittavien väylien ilmentymät ja lohkon sisään määritetään tarvittavia muuttujia DUT:n toimimiseksi.

Usein DUT:n ilmentymälle annettavat parametrit määritetään muuttujiksi erillisessä paketissa, jotta ne voidaan tarvittaessa jakaa testipenkin osille. DUT wrapper -lohkossa tuodaan parametrimuuttujien paketista tarvittava sisältö ja asetetaan muuttujat DUT:n ilmentymän parametreiksi. Lopuksi DUT wrapper -lohkolle argumenteiksi annettujen väylien signaalit liitetään DUT:n ilmentymän portteihin. Testipenkkilohkossa muodostetaan DUT wrapper-lohkosta ilmentymiä ja annetaan niille parametreiksi tarvittavat väylät.

4.9 Sekvenssi alkio

Luokka *uvm_sequence_item* periytyy epäsuorasti *uvm_object*-luokasta luokan *uvm_transaction* kautta ja perii kaikki niiden metodit. Käyttäjä laajentaa *uvm_sequence_item*-luokkaa omalla sekvenssi alkio -luokalla eli luo *uvm_sequence_item*-luokalle aliluokan.

Sekvenssi alkio -luokka kuvaa tietokenttiä, joilla kuljetetaan dataa stimuluksen tuottamiseksi DUT:lle sekä DUT:n ulostulosignaalien arvojen lukemiseksi. Käyttäjän luomassa sekvenssi alkio -luokassa tehdään seuraavat toimenpiteet

- käyttäjän luokka periytetään *uvm_sequence_item*-luokasta
- luokan nimi rekisteröidään tehtaalle *`uvm_object_utils*-makroilla
- luokalle luodaan rakentajafunktio, jonka argumentiksi asetetaan *string name* ja sen sisällä kutsutaan *super.new(name)*
- Luokassa määritellään jäsenmuuttujat, joita tarvitaan tuottamaan stimulusta tai lukemaan DUT:n väylän signaaleille.

Käyttäjä voi myös rekisteröidä *`uvm_field*-makroilla luokan jäsenmuuttujat. Tällöin, käyttäjä voi myöhemmin UVM-komponenteissa kutsua *print*-metodia tulostaakseen sekvenssi alkion jäsenmuuttujien arvot. (IEEE 1800.2-2020 2020, 177–181; UVM_Sequence_item n.d.)

4.10 Sekvenssi

Luokka *uvm_sequence* periytyy epäsuorasti *uvm_sequence_item*-luokasta *uvm_sequence_base*-luokan kautta, joka yhdessä *uvm_sequence*-luokan kanssa sisältää rajapinnat ja metodit sekvenssien tuottamiseen. Uusi sekvenssi luodaan sekvenssiluokan *body*-metodissa, kun jokin UVM-komponentti kutsuu *uvm_sequence*-luokan *start*-metodia.

Käyttäjä laajentaa *uvm_sequence*-luokkaa omalla sekvenssiluokalla, jonka *body*-metodissa käyttäjä määrittelee logiikan sekvenssin luomiselle. Käyttäjän luoma sekvenssiluokka seuraa seuraavia toimenpiteitä

- luokka periytetään *uvm_sequence*-luokasta ja parametrisoidaan sekvenssi alkio -luokalla
- luokan nimi rekisteröidään tehtaalle *`uvm_object_utils*-makrolla
- luokalle luodaan rakentajafunktio, jonka argumentiksi asetetaan *string name* ja sen sisällä kutsutaan *super.new(name)*
- määritetään jäsenmuuttujat, joita tarvitaan ajamaan DUT:lle stimulusta
- määritetään sekvenssi alkio -luokka tyyppinen muuttuja, jolla käyttäjän luoma sekvenssiluokka on parametrisoitu
- määritetään *body*-metodi ja sen logiikka.

Tehtävässä *body* luodaan tehtaan avulla ensin uusi sekvenssi alkio -luokan olio, jolla käyttäjän luoma sekvenssiluokka on parametrisoitu. Olion luonnin jälkeen, *body*-tehtävään voidaan toteuttaa *for*-silmukka, joka tuottaa käyttäjän määrittelemän määrän verran sekvenssi alkioita eli sekvenssi alkio -luokan olioita. Määrä ilmoitetaan *for*-silmukan muuttujalla, joka on yleensä kokonaisluku *i*. Useat sekvenssi alkiot luovat yhdessä DUT:lle ajettavan sekvenssin.

Silmukan sisällä kutsutaan kantaluokan metodeja *start_item* ja *finish_item*. Metodeille annetaan luotu sekvenssi alkio, jonka data voidaan satunnaistaa metodien välissä. Metodi *start_item* lähettää sekvensoijalle sekvenssi alkion ja *finish-_item*-metodilla lopetetaan *body*-metodin suoritus ja talletetaan olio sekvensoijan FIFO-puskuriin. Silmukan ja satunnaistamisen yhdistelmä luo DUT:lle ajettavan sekvenssin, joka sisältää satunnaista dataa. (IEEE 1800.2-2020 2020, 177, 183, 189–190; UVM Sequence [uvm_sequence] n.d.)

Satunnaistaminen ei ole pakollista. Enemmän hallintaa sekvenssi alkion sisällöstä saadaan, kun silmukka jätetään pois ja sekvenssiluokalle määritetään jäsenmuuttujia. Jäsenmuuttujien arvot voidaan sitten asettaa sekvenssi alkion jäsenmuuttujien arvoiksi satunnaistamisen paikalla. Voidaan myös säilyttää silmukka ja määrittää kontrollisignaali, jolla ilmoittaa sekvenssiluokalle, tuleeko data satunnaistaa vai ei. Tapoja luoda sekvenssejä on monenlaisia ja määräytyvät sen mukaan, mitä ja, miten DUT:lle tulisi antaa syötteitä, jotta lohkon toiminta tulisi varmistettua.

4.11 Sekvensoija

Sekvensoijaluokan kantaluokka *uvm_sequencer* periytyy epäsuorasti *uvm_component*-luokasta *uvm_sequencer_base*-luokan kautta ja on eräs UVM-komponenttiluokka, joka tulee parametrisoida käytettävällä sekvenssi alkio -luokalla. Sekvensoija liitetään ajuriin sen kantaluokan sisältämällä *seq_item_export* nimisellä TLM-portti- ja liitäntätyypillä *uvm_seq_item_pull_imp*. Sekvensoija toimii sekvenssi alkioiden välittäjänä sekvenssin ja ajurin välillä sekä luo niiden välille semaforisen sekvenssi alkioiden liikenteen. (IEEE 1800.2-2020 2020, 194–197; Driver Sequencer Handshake n.d.)

Toisin kuin muut komponentit, *uvm_sequencer*-luokasta ei yleensä ole tarpeellista luoda erillistä periytyvää käyttäjän luomaa luokkaa, koska sen perustoiminta on usein riittävä. Käyttäjä voi luoda periytyvän luokan tehdäkseen paremmin selväksi sekvensoijan olemassaolon tai vain nimeämisen takia. Uusi sekvensoija voidaan siis luoda testipenkin ympäristöön myös pelkästään käyttämällä kantaluokkaa. Joissain monimutkaisemmissa toteutuksissa, käyttäjälle saattaa tulla tarve lisätä omaa toiminnallisuutta sekvensoijalle.

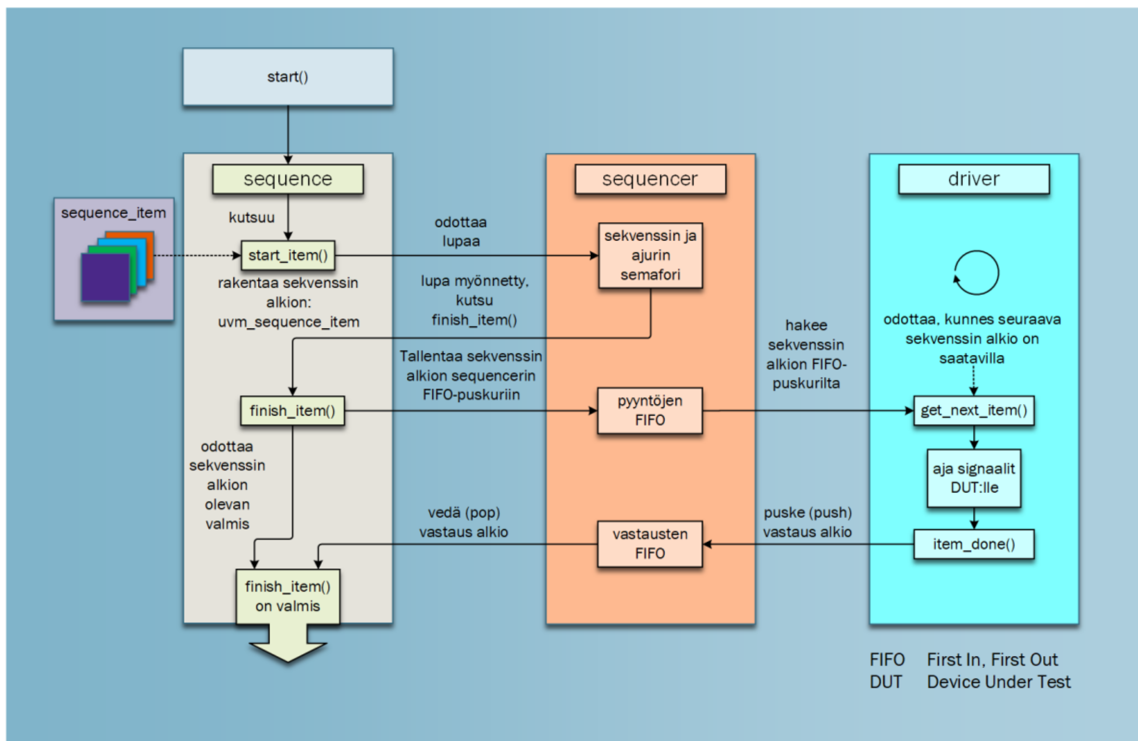
4.12 Ajuri

Ajurin kantaluokka *uvm_driver* periytyy suoraan luokasta *uvm_component*. Luokista *uvm_driver* ja *uvm_sequencer* on olemassa myös vaihtoehtoiset luokat *uvm_push_driver* ja *uvm_push_sequencer*, joiden erona on kommunikointitapa.

Sekvensoija ja ajuri, joiden luokat periytyvät *uvm_sequencer-* ja *uvm_driver*-luokista, kommunikoivat ajurin pyytäessä uutta sekvenssi alkiota, jonka sekvensoija lähettää *finish_item*-metodin kutsun jälkeen. Sekvensoija ja ajuri, joiden luokat periytyvät *uvm_push_sequencer-* ja *uvm_push_driver*-luokista, puolestaan kommunikoivat sekvensoijan lähettämällä uusi sekvenssi alkio ajurille, joka estää uudet transaktiot sekvensoijalta, kunnes on valmis vastaanottamaan seuraavaan.

Aikaisemmassa luvussa esitetyn *uvm_sequencer*-luokan *seq_item_export*-porttityypille on olemassa uvm_*driver*-luokassa vastakappaleena *seq_item_port* niminen *uvm_seq_item_pull_port* TLM-portti- ja liitäntätyyppi. Nämä kaksi porttia yhdistetään luokassa, jossa luodaan sekvensoija ja ajuri eli yleensä agentissa. Vaihtoehtoisille *uvm_push_sequencer*- ja *uvm_push_driver*-luokille on määritelty omat TLM-portti- ja liitäntätyypit. (IEEE 1800.2-2020 2020, 175, 194–195; Driver Sequencer Handshake n.d.)

Sekvensoijan ja ajurin kommunikointi, kun kyseessä on *uvm_sequencer*- ja *uvm_driver*-luokista periytyvät luokat, käynnistyy jonkin komponentin kutsuessa *uvm_sequence*-luokan *start*-funktiota *run_phase*-vaiheen aikana. Funktiokutsu käynnistää sekvenssin metodin *body*, jossa kutsutaan metodit *start_item* ja *finish_item*. Tämän jälkeen sekvensoijan FIFO-puskuriin on talletettu uusi sekvenssi alkio. Lopuksi ajuri suorittaa oman *run_phase*-logiikkansa ja ajaa DUT:n väylän sisääntulosignaaleille uudet arvot. Käyttäjän luomista luokista luotujen sekvensoijan, sekvenssin ja ajurin välinen kommunikointiprosessi käyttäen ajurin *get_next_item*-metodia havainnollistetaan kuviossa 5 (Using get_next_item n.d.)



KUVIO 5. Sekvenssin, sekvensoijan ja ajurin välinen kommunikointiprosessi (Using get_next_item n.d., muokattu)

Komponenttien väliseen kommunikointiin voidaan *get_next_item*-metodin sijan myös käyttää ajurin metodeja *get* ja *put*. Metodit *get* ja *put* luovat käyttäjän kontrolloiman vuorovaikutuksen sekvensoijan ja ajurin välille. Metodilla *get* haetaan uusi sekvenssi alkio ja metodilla *put* lähetetään sekvenssille vastaus takaisin. (Using get() and put() n.d.) Käyttäjän luomassa ajuriluokassa tehdään seuraavat toimenpiteet

- käyttäjän luoma luokka periytetään *uvm_driver*-luokasta
- luokka parametrisoidaan sekvenssi alkio -luokalla
- luokka rekisteröidään `*uvm_component_utils*-makrolla
- luokkaan luodaan virtuaalinen väylän ilmentymä väylästä, jonka signaaleja ajurin tulisi ajaa
- luokkaan luodaan sekvenssi alkio -luokka tyyppinen muuttuja, jolla käyttäjän luoma ajuriluokka on parametrisoitu
- luokalle luodaan rakentajafunktio, jolle asetetaan argumentit *string name* ja *uvm_component* sekä kutsutaan sen sisällä *super.new(name,component)*
- luokalle luodaan *build_phase*-funktio, jossa haetaan *uvm_config_db*-tietokannasta testipenkin top-tasolla asetettu väylä sekä luodaan sekvenssi alkio
- luokalle luodaan *run_phase*-tehtävä, johon luodaan logiikka ajamaan signaaleita DUT:lle riippuen, mitä metodia käytetään hakemaan uusi olio.

Parametrisoidun sekvenssi alkio -luokan on vastattava käytettäviä sekvensoija- ja sekvenssiluokan parametrisointeja, koska eri sekvenssi alkio -luokalla parametrisoituja luokkia ei voi yhdistää keskenänsä. Jokaista erilaisella sekvenssi alkio -luokalla parametrisoitua ajuriluokkaa kohden on oltava siis olemassa samanlaisella sekvenssi alkio -luokalla parametrisoitu sekvensoija- ja sekvenssiluokka, jotta arvoja voidaan ajaa DUT:lle. Parametrisointi voidaan jättää sekvensoijaluokan tapauksessa sen olion luonnin yhteyteen, jos käytetään sen kantaluokkaa.

4.13 Monitori

Monitorin kantaluokkana on *uvm_monitor*, joka periytyy suoraan *uvm_component*-luokasta. Monitorin tarkoituksena on lukea DUT:n väylältä tulevaa dataa ja siirtää se käyttämällä sekvenssi alkiota TLM-porttien kautta muille UVM-komponenteille. Käyttäjän monitoriluokan tulisi toteuttaa seuraavat toimenpiteet

- luokka periytetään *uvm_monitor*-luokasta
- luokka rekisteröidään tehtaalle *`uvm_component_utils*-makrolla
- luokkaan luodaan virtuaalinen väylä siitä väylästä, jonka signaaleja monitorin tulisi tarkastella
- luokkaan luodaan muuttuja tyypiltä sekvenssi alkio -luokka
- luokkaan luodaan TLM-portin ilmentymä sekvenssi alkioiden siirtämiseksi
- luokalle luodaan rakentajafunktio, jolle asetetaan argumentit *string name* ja *uvm_component* sekä kutsutaan sen sisällä *super.new(name,component)*
- luokkaan luodaan *build_phase*-funktio, jossa haetaan *uvm_config_db*-tietokannasta testipenkin top-tasolla asetettu väylä ja luodaan sekvenssi alkio sekä monitorin TLM-portti
- luokkaan luodaan *run_phase*-metodi, joka sisältää monitoroinnin logiikan.

Metodiin *run_phase* luodaan *forever*-silmukka, jossa määritellään @-käskyllä tapahtuman hetki, kun dataa väylältä pitäisi siirtää muille komponenteille. Tapahtuman toteutuessa, väylän signaalien arvot tai metodien kutsut asetetaan sekvenssi alkion jäsenmuuttujien arvoiksi ja lopuksi kutsutaan silmukan sisällä TLM-portin metodia *write*, jolle annetaan argumentiksi sekvenssi alkio. (UVM Monitor [uvm_monitor] n.d.)

Tapahtumia voi olla useampia, jolloin silmukan sisään voidaan luoda *fork-join_any*-lohko, jonka sisään määritetään useampi tapahtumaprosessi @-käskyillä. Tapahtumaprosesseissa on tarkoituksena liipaista, kuten oskilloskoopissa, eri signaaleihin ja siirtää uudet arvot sen jälkeen käyttämällä sekvenssi alkioita.

4.14 Agentti

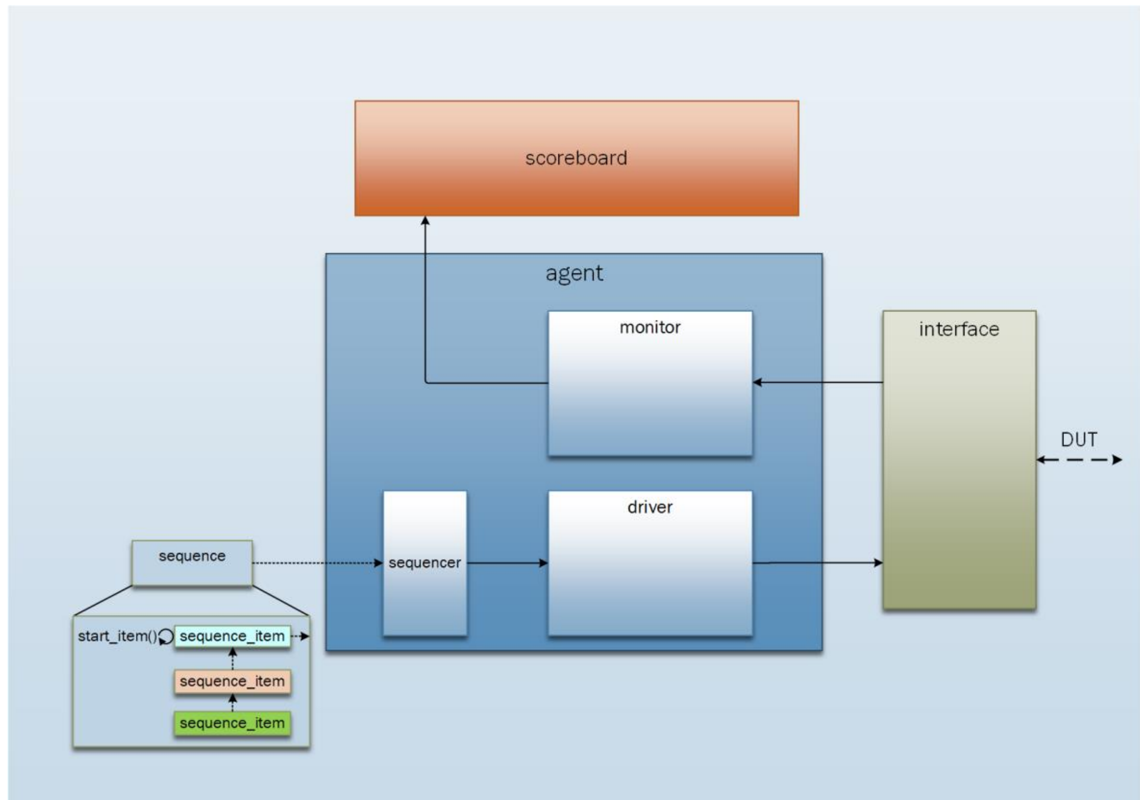
Agentin kantaluokka *uvm_agent* periytyy suoraan *uvm_component*-luokasta ja käyttäjän luoman agenttiluokka tulisi periytyä *uvm_agent*-kantaluokasta. Agenttiluokan tarkoituksena on kapseloida sekvensoija, ajuri ja monitori sekä liittää niiden TLM-porttityypit.

Käyttäjän luomaa agenttia kutsutaan aktiiviseksi, jos sen kaikki kolme komponenttia ovat käytössä ja passiiviseksi, jos ainoastaan monitori on käytössä. Käyttäjä luo *uvm_agent*-luokasta periytyvän agenttiluokan seuraavilla toimenpiteillä

- luokka periytetään *uvm_agent*-luokasta
- luokka rekisteröidään *`uvm_component_utils*-makrolla UVM-tehtaalle
- luokkaan luodaan sekvensoija-, ajuri-, ja monitoriluokka tyyppiset muuttujat
- luokkaan luodaan TLM-portti, joka liitetään monitorin TLM-porttiin, siirtääkseen dataa tulostaululle
- luokalle luodaan rakentajafunktio, jolle asetetaan argumentit *string name* ja *uvm_component* sekä kutsutaan sen sisällä *super.new(name,component)*
- luokalle luodaan metodi *build_phase*, jossa oliot luodaan niille tarkoitettuihin kahvoihin käyttämällä UVM-tehtaan metodeja sekä voidaan tehdä ehtolauseet konfigurointia varten
- luokkaan luodaan metodi *connect_phase*, jossa sekvensoijan ja ajurin sekä monitorin ja agentin TLM-porttityypit liitetään.

Agenttiluokalle voidaan myös asettaa komponenttien luomisen yhteyteen ehtolauseita konfigurointia varten. Konfigurointi toteutetaan luomalla konfigurointiluokka, jonka olio luodaan agenttiluokkaan. Konfigurointiluokassa voidaan määritellä siten esimerkiksi, onko agentti aktiivinen vai ei. (UVM Agent | uvm_agent n.d.)

Agentti siis sisältää kaikki komponentit, joita tarvitaan vuorovaikuttamaan DUT:n kanssa. Kuviossa 6 visualisoidaan, miltä agentti näyttää, kun eri komponentit visualisoidaan lohkoina. Esitetyt lohkot kuvaavat siis luotuja olioita ja väyliä toisten luokkien tai moduulien sisällä.



KUVIO 6. Agentin visualisointi (UVM Agent | uvm_agent n.d., muokattu)

4.15 Tulostaulu

Luokka *uvm_scoreboard* periytyy suoraan *uvm_component*-luokasta. Tulostaulun tarkoituksena on verrata DUT:n ulostuloja odotettuihin arvoihin eli sen vastuulla on laitteistokuvauksen verifiointiin käytetty logiikka. Luokka vastaanottaa TLM-porttityyppien kautta tulevia sekvenssi alkioita yleensä monitorilta ja vertaa vastaanotettua dataa odotettuihin arvoihin.

Jokainen tulostaululuokka on erilainen, koska jokainen DUT tarvitsee verifioida eri tavoin ja tulostaululuokan *run_phase*-vaiheen logiikka on siksi aina ainutlaatuinen. Jokainen käyttäjän luoma tulostaululuokka seuraa kuitenkin seuraavia toimenpiteitä

- luokka periytetään *uvm_scoreboard*-luokasta
- luokka rekisteröidään UVM-tehtaalle käyttäen `*uvm_component_utils*-makroa

- luokkaan luodaan tarvittavat TLM-porttityyppien ilmentymät, jotka yleensä ovat porttityypiltään *imp* ja käytetään monitorin TLM-portteja varten
- luokalle luodaan rakentajafunktio, jolle asetetaan argumentit *string name* ja *uvm_component* sekä kutsutaan sen sisällä *super.new(name,component)*
- luokalle luodaan *imp*-porttityypeille *write*-metodit
- luokassa määritellään, mitä saadulle sekvenssi alkioiden sisältämälle datalle tehdään, yleensä *imp*-porttityypille luodussa *write*-metodissa
- luokalle luodaan *build_phase*-metodi, jossa luodaan TLM-porttityypit *new*-metodilla
- luokkaan luodaan logiikka, joka tarkistaa ulostulojen oikeellisuuden odotettuja arvoja vastaan.

Itse verifioinnin logiikka toteutetaan luokan *run_phase*-metodissa, jos tarkoituksena on koko simuloinnin ajan ajaa tarkistuksia. Tarkistukset voidaan myös toteuttaa ilman simulointiajan askeltamista *check_phase*-vaiheessa. (UVM Scoreboard n.d.)

TulostauluTLM-porttityypit luodaan usein *imp*-porttityypeiksi, koska tulevan datan päätepisteenä on yleensä tulostaulu. Testipenkin sisältäessä useita agentteja ja tulostaulun sisältäessä siis useita *analysis_imp*-portteja, *write*-funktion kanssa tulee ongelmaksi tunnistaa, minkä portin *write*-funktio on kyseessä. Tähän on avuksi standardissa IEEE 1800.2-2020 (2020) määritetty *`uvm_analysis_imp_decl*-makro, jolla voidaan luoda *analysis_imp*-porteille omat tunnisteet *write*-funktiota varten.

Makrolle *`uvm_analysis_imp_decl* annetaan argumentiksi haluttu loppuliite. (IEEE 1800.2-2020 2020, 390.) Esimerkiksi makrolla voitaisiin luoda loppuliite *_protokolla*, jolloin *write*-funktio tulisikin luoda tulostaululuokkaan nimellä *write_protokolla*. Samoin luodessa uutta porttityypin ilmentymää, esimerkiksi *uvm_analysis_imp*, tulisikin portin tyypiksi *uvm_analysis_imp_protokolla*, jonka perään määritettäisiin haluttu portin ilmentymän nimi. (TLM Implementation Port Declaration Macros n.d.)

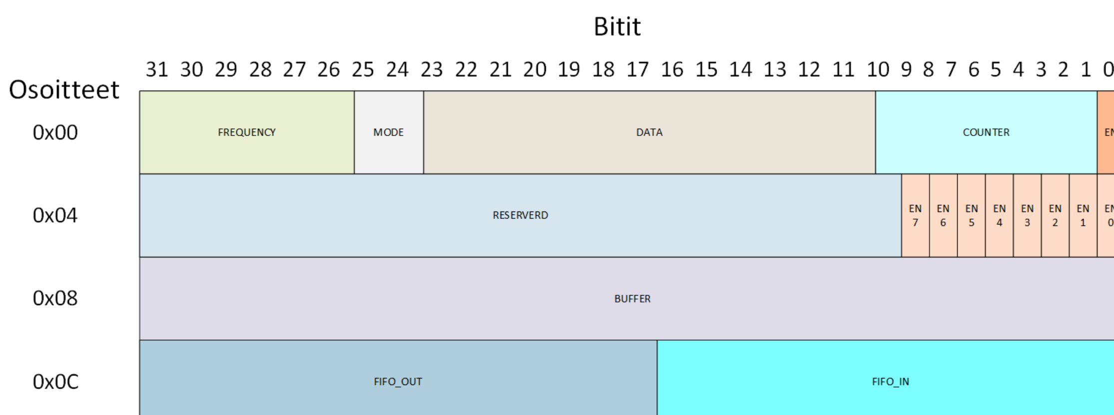
4.16 Rekistereiden kanssa vuorovaikuttaminen

Digitaalipiirilohkot saattavat sisältää ohjelmoitavia rekistereitä ja muisteja, joita voidaan lukea ja kirjoittaa käyttämällä protokollia. Esimerkiksi voisi olla 32-bittinen rekisteri, jonka bitit ovat vielä jaoteltu eri kokoisiin bittijoukkoihin ja voivat esittää kontrollisignaaleja tai jotain toiminnallisuutta.

Kyseisten rekistereiden avulla voidaan näin ohjelmoimalla konfiguroida piiri myös sen valmistuksen jälkeen. Useat rekisterit kootaan yhdeksi lohkoksi, jota kutsutaan rekisterilohkoksi. (Register Layer n.d.)

4.16.1 Rekisterilohkot

Rekisterilohkossa annetaan jokaiselle rekisterille oma numeerinen osoitteensa, joiden osoitejako riippuu rekistereiden koosta eli niiden bittimäärästä. Esimerkiksi 32-bittisten rekistereiden osoitteet ovat jaoteltu neljän tavun välein, koska 32-bittiä vastaa neljää tavua eli ensimmäisen rekisterin osoite olisi heksadesimaalina 0x00 ja seuraava osoite olisi 0x04. Kuviossa 7 nähdään, miltä rekisterilohko näyttäisi.



KUVIO 7. Rekisterilohko (Register Layer n.d., muokattu)

Järjestelmäpiiri koostuu useista osalohkoista, jotka saattavat sisältää jokainen oman rekisterilohkonsa ja eri osalohkot voidaan kategorisoida eri ryhmiin kuten esimerkiksi I/O-laitteiden osalohkot. Kommunikoidakseen eri lohkoryhmien osalohkojen rekistereiden kanssa, järjestelmäpiiriin luodaan muistikartta, jolla on

rekisterilohkon tapaan osoitteista koostuva osoiteavaruus. Muistikartan osoitteet viittaavat eri lohkoryhmien osalohkoihin, joille on annettu omat osoitteet. Muistikartan osoiteavaruus jaetaan sitten muistialueisiin, jotka alkavat lohkoryhmän ensimmäisen osalohkon osoitteesta ja päättyvät viimeisen lohkoryhmän osalohkon osoitteeseen.

Viitatakseen tietyn lohkoryhmän osalohkon rekisterilohkon rekisteriin, osoitteeksi kirjoitetaan osalohkon osoite + rekisterin osoite. Tällöin osoite ensin tulkitaan muistikartan avulla tietyn osalohkon osoitteeksi, jonka jälkeen osalohko itse tulkitsee osoitteesta lisätyn poikkeaman (eng. offset) eli rekisterin osoitteen. (Register Layer n.d.)

4.16.2 Rekisterirajapinta

UVM sisältää rekisterirajapintaluokkia, joilla mahdollistetaan DUT:n rekisterilohkojen rekistereiden lukeminen ja niihin kirjoittaminen. Rekisterirajapintaluokat luovat yhdessä rekisterirajapinnan eli RALin (Register Access Layer), joka luo korkeamman tason esityksen DUT:n kartoitetusta muistialueesta ja rekistereistä. RAL siis mallintaa DUT:n rekisterilohkoa luokkina ja olioina, joita käytetään sitten vuorovaikuttamaan DUT:n rekistereiden kanssa.

RALin rekisterit sisältävät kaksi arvoa: toivottu ja peilattu arvo. Rekisterin toivottu arvo on se data, joka kirjoitetaan DUT:n rekisteriin kutsuessa *update*-metodia ja peilattu arvo puolestaan peilaa DUT:n rekisterissä olevaa arvoa, kun kutsutaan *read*-metodia.

Käyttäjä luo RALin rekisterin luomalla rekisteriluokan, joka periytyy *uvm_reg*-luokasta. RALin rekisteriluokan sisällä luodaan rekisterin jokaista bittijoukkoa varten tyyppiä *uvm_reg_field*-luokka olevat muuttujat, joille annetaan bittijoukon nimi. Rekisteriluokalle luodaan sitten sen rakentajafunktio *new* ja kutsutaan sen sisällä *uvm_reg*-luokan rakentajafunktiota. Luokan *uvm_reg* rakentajafunktiolle on annettava argumenteiksi rekisterin nimi, rekisterin koko sekä rekisterin toiminnallisen kattavuuden malli, jos sellaista käytetään, tässä järjestyksessä. Esimerkiksi

new(enable, 32, UVM_NO_COVERAGE) loisi 32-bittisen *enable*-rekisterin ilman toiminnallisen kattavuuden mallia.

Rakentajafunktion lisäksi on luotava *build*-metodi, jossa rekisterin eri bittijoukkoja vastaavat oliot luodaan ja konfiguroidaan. Jokainen bittijoukkoa esittävä olio luodaan tehtaan avulla asettamalla niitä vastaavien olioiden kahvojen arvoksi *uvm_reg_field::type_id::create*-käsky. Bittijoukkoja vastaavien olioiden konfiguroinnilla tarkoitetaan *uvm_reg_field*-luokan *configure*-metodin kutsua, joka sisältää jäsenmuuttujat esimerkiksi bittijoukon koolle tai sen pääsyoikeudelle. Jokaista DUT:n rekisteriä varten luodaan oma rekisteriluokka ja tehdään edellä kuvatut toimenpiteet.

Rekisteriluokkien luomisen jälkeen luodaan rekisterilohkoluokka, jonka sisään luodaan tyyppiä rekisteriluokka olevat muuttujat. Muuttujat kuvaavat jokaista rekisterilohkon rekisteriä. Käyttäjän luoma rekisterilohkoluokan on periydyttävä luokasta *uvm_reg_block*. Rekisterilohkoluokkaan luodaan lisäksi metodi *build*, jossa

- luodaan muistikartta
- luodaan rekisteriluokkien oliot
- konfiguroidaan rekisteriluokkien oliot
- kutsutaan rekisteriluokkien *build*-metodia
- lisätään rekisteriluokkien oliot muistikarttaan.

Luokasta uvm*_reg_block* periytyvästä käyttäjän luomasta luokasta luotu olio on rekisterilohkon malli, jota käytetään rajapintana vuorovaikuttamaan DUT:n rekistereiden kanssa eli RAL. RAL siis koostuu kolmesta eri luokasta: *uvm_reg, uvm_reg_field* ja *uvm_reg_block*, joita käyttäjä laajentaa kuvaamaan omaa rekisterilohkoonsa. (UVM Register Model n.d.)

4.16.3 Adapteri ja syötteenennustaja

RALin lisäksi on kolme muuta komponenttia, joita tarvitaan tekemään rekisterikirjoituksia ja -lukuja. Tarvittavat komponentit ovat

- Protokolla-agentti
- adapteri
- syötteenennustaja.

Adapteri muuttaa RALin siirtämien olioiden data protokollaspesifiseksi protokolla-agentille, jonka tarkoituksena on sitten kommunikoida DUT:n rekistereiden kanssa tietyn protokollan mukaisesti. RALin *uvm_reg*-luokalla on *read*- ja *write*-metodit, joilla rekistereihin kirjoitetaan tai niistä luetaan dataa. Metodeille *read* ja *write* annetaan argumenteiksi tyyppiä *uvm_status_e* oleva muuttuja *status* sekä rekisterille menevä data, jonka jälkeen RAL kommunikoi käyttämällä *uvm_reg_bus_op*-tyyppisiä syötteitä, jotka sisältävät seuraavat tiedot

- *kind*, joka voi olla READ tai WRITE ja kuvaa kirjoitus- tai lukutapahtumaa
- *addr* kuvaa kohdeosoitetta rekisterissä
- *data* kuvaa kirjoitettavaa tai luettavaa dataa
- *n_bits* kuvaa *uvm_reg_item::value* bittien määrää, jotka siirretään
- *status* kuvaa transaktion tulosta, joka voi olla UVM_IS_OK, UVM_HAS_X tai UVM_NOT_OK.

Adaptereiden kantaluokkakirjastona toimii *uvm_reg_adapter*, joka periytyy *uvm_object*-luokasta. Luokka *uvm_reg_adapter* sisältää metodit *reg2bus* ja *bus2reg*, jotka käyttäjän tulee luoda omaan luokkaansa muuttamaan *uvm_reg_bus_op*-tyyppiset syötteet protokollaspesifiseksi ja toisinpäin.
(UVM RAL Adapter n.d.; UVM Register Environment n.d.; IEEE 1800.2-2020 2020, 350–358.)

Syötteenennustajan tarkoituksena on puolestaan lukea agentin monitorilta saatuja syötteitä ja päivittää RAL vastaamaan DUT:n rekistereiden dataa. Syötteenennustaja voi olla implisiittinen tai eksplisiittinen. Implisiittinen syötteenennustaja päivittää tai peilaa DUT:n rekistereiden arvot automaattisesti jokaisen kirjoitus- tai lukuoperaation jälkeen. Eksplisiittinen toteutus asettaa implisiittisen

syötteenennustamisen pois päältä ja jokainen rekisterin peilaus tehdään ulkoisesti käyttämällä *uvm_reg_predictor*-kantaluokan aliluokan komponenttia. (UVM RAL Predictor n.d.; UVM Register Environment n.d.)

4.16.4 Rekisterirajapinnan ympäristö

Rekisterirajapinnalle, adapterille ja syötteenennustajalle voidaan luoda oma ympäristö, joka periytyy *uvm_env*-luokasta tai se voidaan luoda suoraan testipenkin ympäristöön. Rekisterirajapinnan oma ympäristö saatetaan luoda tehdäkseen testipenkistä paremmin uudelleen käytettävän ja kapseloimaan komponentit sisäänsä. Molemmissa tapauksissa on kyse kuitenkin näiden kolmen komponentin liittämisestä ja luomisesta ympäristöön.

Ympäristöön siis luodaan protokolla-agentti, RAL, adapteri sekä mahdollinen eksplisiittinen syötteenennustaja. Ympäristöluokkaan määritellään ensin jokaisesta luokasta muuttujat sisältämään olioiden kahvoja, johon luodaan niille oliot *build_phase*-metodissa. Olioiden luonnin lisäksi jokainen komponentti on vielä liitettävä käyttäjän luomassa *connect_phase*-metodissa. Toimenpiteet, jotka käyttäjän tulee tehdä luodakseen rekisterirajapinnalle ympäristön ovat

- käyttäjän luoma ympäristöluokka periytetään *uvm_env*-kantaluokasta
- luokka rekisteröidään tehtaalle `*uvm_component_utils*-makrolla
- komponenteille luodaan tyypeiltään niiden luokkia olevat muuttujat
- luokalle luodaan rakentajafunktio, jolle asetetaan argumentit *string name* ja *uvm_component* sekä kutsutaan sen sisällä *super.new(name,component)*
- komponenteille luodaan oliot niitä vastaaviin kahvoihin käyttäjän luomassa *build_phase*-metodissa UVM-tehtaan avulla
- Olioiden luomisen lisäksi *build_phase*-metodissa kutsutaan RALin metodit *build*, *lock_model* ja *reset* rakentamaan, lukitsemaan ja uudelleenkäynnistämään RALin
- *build_phase*-metodissa asetetaan RALin olion kahva *uvm_config_db*-tietokantaan, jotta se voidaan hakea testiluokassa
- liitetään protokolla-agentin sisältämä monitori tulostauluun tai syötteenennustajaan *connect_phase*-metodissa

- liitetään protokolla-agentin sekvensoija sekä adapteri RALin muistikartalle *set_sequencer*-metodilla.

Usein oletuksena käytetään implisiittistä syötteenennustajaa, joka päivittää jokaiselle luku- tai kirjoitusjaksolla RALin vastaamaan DUT:n rekisterilohkoa. Ulkoista eli eksplisiittistä syötteenennustajaa voi kuitenkin käyttää halutessaan tai tapauksessa, jossa jokaisen kirjoitus- tai lukusekvenssin jälkeen ei haluta päivittää suoraan RALia. (UVM Register Environment n.d.; Connecting register env n.d.)

Syötteenennustaja on ainoa komponentti, joka saattaa muuttua toteutuksesta toiseen ja sen käyttö riippuu toteutuksesta. Luotiin rekisterirajapinta ympäristöön tai omaan ympäristöönsä, tärkeintä on olla sen luomisessa johdonmukainen myös muiden projektien kanssa.

Osa ympäristöstä saattaa olla jo luotuna ja kapseloituna käyttäessä VIPejä (Verification IP). Parhaimmassa tapauksessa VIP on niin yksinkertainen käyttää, että kaikki edellä kuvatut toimenpiteet tehtäisiin jo valmiiksi jossain VIPin luokassa ja käyttäjälle jää ainoastaan olion luominen testipenkin ympäristöön. VIPien merkitys tuleekin usein ilmi kehittäessä testipenkkiin ympäristöä, jossa DUT:lle tulisi syöttää standardoidulla protokollalla syötteitä ja huomataan, kuinka paljon VIP yksinkertaistaa ja nopeuttaa testipenkin luomista.

4.17 Ympäristö

UVM ympäristö on itse testipenkin ympäristö, jossa luodaan ja liitetään jokainen testiympäristön UVM-komponentti. Ympäristön kantaluokkana toimii *uvm_env*, joka periytyy suoraan *uvm_component*-luokasta. (IEEE 1800.2-2020 2020, 174.)

Ympäristössä luodaan myös uudelleenkäytettävät verifiointi komponentit eli VIPien luokkien ilmentymät sekä mahdolliset konfiguraatioehtolauseet ympäristön konfiguroimiseksi. Testipenkissä voisi luoda nämä kaikki komponentit myös testiluokassa, mutta ympäristön käyttäminen UVM:n metodologian lisäksi tuo tietyt

hyödyt esimerkiksi ympäristö voidaan konfiguroida jokaista testiä varten eri tavalla. Käyttäjän ympäristön luomisen tulisi noudattaa seuraavia toimenpiteitä

- luokka periytyy *uvm_env*-kantaluokasta
- luokka rekisteröidään tehtaalle käyttäen *´uvm_component_utils*-makroa
- luokkaan luodaan muuttujat agenttien, rekisterirajapinnan ympäristön ja tulostaulun sekä mahdollisesti muiden luokkien olioiden kahvojen säilyttämiseksi kuten RALin, adapterin ja protokolla-agentin, jos rekisterirajapinnan ympäristöä ei käytetä
- ympäristöön luodussa *build_phase*-metodissa luodaan kaikki tarvittavat oliot ympäristöön käyttämällä UVM-tehtaan metodeja
- ympäristöön luodussa *connect_phase*-metodissa liitetään kaikki tarvittavat TLM-porttityypit kuten agenttien TLM-porttityypit tulostaulun TLM-porttityyppiin

Luotua ympäristöä voidaan myös uudelleen käyttää korkeamman hierarkian lohkon verifioimiseksi. Ympäristö luokasta ei tarvitse muuta luoda kuin olio toiseen ympäristöön ja konfiguroida haluamakseen. (UVM Environment [uvm_env] n.d.)

4.18 Testi

Luokka *uvm_test* on kantaluokka, josta käyttäjän omat testiluokat tulisi periytyä. Testiluokkien tulisi periytyä *uvm_test*-kantaluokasta, jotta käyttäjän olisi mahdollista valita suoritettava testi komentoriviltä UVM_TESTNAME-komennolla tai antamalla testin nimi *run_test*-metodille. (IEEE 1800.2-2020 2020, 172–173, 452.)

Testi on yksinkertaisuudessaan sarja käskyjä, joilla tuottaa erilaisia sekvenssejä tai syötteitä DUT:lle, joiden vaikutusta tulostaulu tarkistaa. Testi käynnistetään kutsumalla *run_test*-metodia, jolle annetaan argumentiksi testin nimi. Metodin argumentit voivat olla myös tyhjiä, jolloin testin nimi on annettava komentoriviltä UVM_TESTNAME-komennolla. Testiluokkaan tehdään seuraavat toimenpiteet

- luokka periytyy *uvm_test*-kantaluokasta
- luokka rekisteröidään tehtaalle *´uvm_component_utils*-makrolla
- luokalle luodaan rakentajafunktio *new*, jolle annetaan nimeksi luokan nimi lainausmerkeissä ja asetetaan *parent*-muuttujan arvoksi *null*

- luokkaan luodaan *build_phase*-metodi, jossa luodaan oliot tai tehdään muut komponenttien rakennusvaiheessa asiat
- luokkaan luodaan sarja käskyjä *run_phase*-metodissa tai sen sisältämiin muihin vaihemetodeihin, joilla verifioidaan DUT
- metodissa *run_phase* kutsutaan alussa *phase.raise_objection(this)* ja lopussa *phase.drop_objection(this)* ilmoittamaan muille komponenteille, koska testi alkaa ja loppuu

Testien käynnistämiseen suositaan komentoriviltä annettua testin nimeä, koska se mahdollistaa joustavasti eri testien valitsemisen. Testin käynnistyksen jälkeen, testiluokassa luodaan koko testipenkin rakenne käyttämällä UVM-komponenttiluokkien olioita. Tämä onnistuu, käyttäessä ympäristöluokkaa kapseloimaan muut komponentit, pelkästään luomalla ympäristöluokasta olion. (UVM Test n.d.)

Jokainen testi on erilainen ja luodaan yleensä tiettyä ominaisuuden verifiointia varten. Tapoja testien luomiselle on monenlaisia, mutta niiden fundamentaalinen idea on sisältää käskyt ajamaan erilaisia syötteitä DUT:lle sen verifioimiseksi. Esimerkiksi testi voisi olla jokaisen rekisterin kirjoittaminen ja lukeminen, jolla verifioitaisiin rekistereiden luku- ja kirjoitusoikeudet.

Rekistereiden kirjoittamiseen tai lukemiseen käytetään testipenkkiluokassa luotua RALin olion *write*- ja *read*-metodeja. RALin käytön yksinkertaistamiseksi on hyvä asettaa sen olion kahva rekisterirajapinnan ympäristössä *uvm_config_db*-tietokantaan ja hakea se testiluokassa. Testiluokassa voidaan sitten helposti hakea ympäristön RALin olion kahva tietokannasta ja käyttää sitä ilman koko hierarkkista polkua.

Sekvenssien luomiseksi DUT:n sisääntuloille puolestaan luodaan sekvenssiluokan olio ja kutsutaan testissä sen *start*-metodia. Sekvenssi voidaan muodostaa monella eri tavalla, mutta suositumpi tapa on yleensä generoida useampia sekvenssi alkioita, joiden data satunnaistetaan sekvenssiluokassa. Kontrolloiduissa sekvensseissä asetetaan testiluokassa sekvenssin jäsenmuuttujien arvot ennen *start*-metodin kutsua.

Käyttäjän täytyy myös osata kontrolloida sekvenssien ja rekisteriluku tai -kirjoituksien ajoittamista, jotta testi kuvastaa mahdollisimman hyvin todellista käyttöympäristöä. Liian nopea sekvenssien syöttäminen tavalla, joka ei kuvasta todellista käyttötilannetta on turhaa, jos DUT ei ole siihen tarkoitettu. Testien tulisikin heijastaa verifiointi- ja suunnitteluspesifikaatioita ja todellisia tilanteita.

5 TESTIPENKIN SUUNNITTELU JA TOTEUTUS

Toteutuksessa käytetään teoriaosiossa esiteltyjä menetelmiä pohjana toteuttamaan UVM-testipenkki. Suojellakseen yrityksen salassa pidettäviä IP-lohkoja ja VIPejä, testipenkin toteutus esitetään siten, että lähdekoodeissa käytetyt nimet abstrahoidaan toisiksi ja lähdekoodeja, jotka voisivat paljastaa jotain salassa pidettävää, ei esitetä. Toteutuksesta tulisi kuitenkin saada käsitys, kuinka testipenkki toteutettiin.

5.1 Suunnittelu

Ennen testipenkin aloittamista suunniteltiin, kuinka testipenkki tultaisiin toteuttamaan. Ensin tutkittiin mahdollisia VIPejä, joita voisi hyödyntää testipenkin toteutuksessa. DUT:n rekistereiden kanssa vuorovaikuttamiseksi käytettävälle AHB-Lite protokollalle oli olemassa VIP, jota voitiin käyttää testipenkissä. AHB-Lite protokolla VIPin lisäksi käytössä tulisi olemaan yrityksen sisäinen työkalu, jolla luoda RAL automaattisesti rekisterikuvauksesta.

Muut testipenkin osat olisi toteuttava itse. Ymmärtääkseen, minkälaiset väylät sekä agentit olisi luotava testipenkille, alettiin tutkimaan verifioitavan lohkon sisältämiä sisään- ja ulostuloportteja. Porteista piti selvittää niiden vaikutus ja suunnitella, kuinka sekvenssejä tulisi testipenkissä luoda. Näiden perusasioiden pohjalta voitiin jo alkaa toteuttamaan testipenkkiä, mutta myöhemmin olisi vielä selvitettävä DUT:n toiminta kokonaisuudessaan tulostaulua ja testejä varten sekä olisi luotava spesifikaatiot. Kuviossa 8 on hahmotelma suunniteltavasta testipenkistä.

KUVIO 8. Suunniteltava testipenkki

5.2 Simulaatioympäristön valmistelut

Työssä käytettiin yrityksen valmista Makefile-tiedostoa, joka sisälsi tarvittavat simulaattorikomennot lähdekoodien kääntämiseksi ja simuloimiseksi. Simulaatioiden ajamista ei optimoitu sen enempää, koska simulaattoreiden sisältämien lukuisten toimintojen ja niihin liittyviä valintakytkimien opettelu olisi jo oma työnsä. Makefile-tiedoston sisältö pyrittiin alusta saakka toteuttamaan ilman tarvetta jälkimuutoksille, mutta muokattiin tarpeen tullen testipenkin muuttuessa. Makefile-tiedostosta saatettiin esimerkiksi poistaa osa käännettävistä testipenkin lähdekoodeista väliaikaisesti. Tämän avulla voitiin rajata virheet virheilmoitusten sattuessa yhteen virheellisen lähdekooditiedostoon, joka tuotti useita virheilmoituksia muissakin lähdekoodeissa.

5.3 Alustava testipenkki

Simulaatioympäristön valmistelujen jälkeen alettiin toteuttamaan itse testipenkkiä. Alkuun luotiin alustava UVM-testipenkki, joka sisälsi ainoastaan testipenkkilohkon, yhden testiluokan ja paketin testiluokille. Alustavan testipenkin tarkoituksena oli varmistaa Makefile-tiedoston toiminta ennen kuin testipenkin lähdekoodeja ehtisi sen enempää kirjoittamaan. Samalla tarkistettiin, että jo olemassa olevat tiedostot kääntyivät.

Testipenkkilohkolle (kuva 14) tuotiin *uvm_pkg.sv*-tiedosto kutsumalla *import uvm_pkg::** eli tuomalla koko paketin sisältö käyttääkseen UVM:n kantaluokkia. Paketin tuonti tehtiin lohkon sisällä välttääkseen globaalin näkyvyysalueen käyttämistä. Paketin *uvm_pk.sv* lisäksi tuotiin testiluokille luotavan paketin koko sisältö sekä sisällytettiin *uvm_macros.svh*-tiedosto *`include*-kääntäjädirektiivillä. Metodille *run_test* annettiin tässä vaiheessa testin nimi lainausmerkkien sisällä komentorivikäskyn sijan.

```
1  module testbench_top;
2    `include "uvm_macros.svh"
3    import uvm_pkg::*;
4    import test_pkg::*;
5
6    initial begin
7      run_test("test");   // Start UVM
8    end
9  endmodule : testbench_top
10
```

KUVA 14. Alustava testipenkkilohko

Testiluokkaan (kuva 15) luotiin yksinkertainen ”Hello World!” tulostus *`umv_info*-makrolla *run_phase*-metodissa. Testiluokan nimi, joka annettiin *run_test*-metodille testipenkkilohkossa, täytyi myös olla rekisteröitynä UVM-tehtaalle, jotta UVM pystyi löytämään sen kutsuessa metodia.

```
`ifndef TEST_SVH_
`define TEST_SVH_

class test extends uvm_test;
  // Register the class to the UVM component factory
  `uvm_component_utils(test)
  `timescale 1ns/1ps

  // Method prototypes
  extern function
      new(string name = "test", uvm_component parent);  // Class constructor
  extern virtual task
      run_phase(uvm_phase phase);                        // Run phase
endclass : test

// Function: new
//
// - Creates test objects
function test::new(string name = "test", uvm_component parent);
  super.new(name,parent);
endfunction : new

// Virtual task: run_phase
//
// TODO this is used to construct a working TB
// - Prints Hello World!
task test::run_phase(uvm_phase phase);
  phase.raise_objection(this);

  `uvm_info ("test::run_phase","Hello World!",UVM_LOW);

  phase.drop_objection(this);
endtask : run_phase

`endif  // TEST_SVH_
```

KUVA 15. Alustava testiluokka

Luodulle testiluokalle sekä tulevia testiluokkia varten luotiin paketti (kuva 16). Testiluokkien paketin sisällä tuotiin *uvm_pkg*-paketin koko sisältö sekä *uvm_macros.svh*-tiedosto sisällyttämällä se `*include*-kääntäjädirektiivillä pakettien eksplisiittisen näkyvyysalueen vuoksi. UVM:n paketin ja makrojen sisällyttämisen lisäksi sisällytettiin vielä luotu testiluokka `*include*-kääntäjädirektiivillä.

```
package test_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  // Include project-specific class headers
  `include "test.svh"
endpackage : test_pkg
```

KUVA 16. Testiluokkien paketti

Seuraavaksi kokeiltiin saada alustavan testipenkin lähdekoodit kääntymään ja korjattiin kääntäjän tulostamat virheet. Tiedostojen onnistuneen käännöksen sekä Makefile-tiedoston toiminnan varmistamisen jälkeen siirryttiin tekemään DUT wrapperia ja pakettia DUT:n parametreille. Kuvassa 17 on onnistuneen alustavan UVM-testipenkin simulaatiotulos ilman graafista käyttöliittymää.

KUVA 17. Simulaatiotulos alustavasta testipenkistä

5.4 Top-taso

Testipenkin top-tason osia olivat testipenkkilohko, paketit, väylät sekä DUT wrapper -lohko. Top-tason lähdekoodit luotiin ensin, koska niiden välillä ei ollut riippuvuuksia ja ne tarvittiin muun testipenkin toteuttamiseksi.

Paketeista luotiin kuitenkin ainoastaan parametrien paketti tässä vaiheessa ja loput paketit luotiin kehittäessä testipenkin luokkia. Top-tasolla luotiin myös itse kellon generoiminen yhdessä väylän kuvauksessa.

5.4.1 DUT wrapper

DUT wrapper -lohkoa varten luotiin paketti, joka sisälsi DUT:n parametrien arvoiksi asetettavat parametrimuuttujat. Pakettiin (kuva 18) listattiin jokaista parametria varten vakiomuuttuja sekä luotiin muuttuja käytettävälle kellon jaksonajalle. Kellon jaksonaikaa oli tarkoitus käyttää myöhemmin testipenkissä generoimaan kelloa.

```
1  package dut_parameters;
2    // Used parameters in the DUT
3    parameter CLK_CYCLE          = 50ns;  // 1/50ns = 20 MHz
4    parameter PARAMETER_1        = 4;
5    parameter PARAMETER_2        = 8;
6    parameter PARAMETER_3        = 5;
7    parameter PARAMETER_4        = 5;
8    parameter PARAMETER_5        = 4;
9    parameter PARAMETER_6        = 20;
10   parameter PARAMETER_7        = 8;
11   parameter PARAMETER_8        = 7;
12 endpackage: dut_parameters
13
```

KUVA 18. Parametrimuuttujien paketti

Testipenkkiä varten luotiin DUT wrapper -lohko (kuva 19), jonne muodostettiin DUT:sta ilmentymä. Lohkolle tuotiin edellä luodun parametrimuuttujien paketin koko sisältö ja ilmentymälle kopioitiin laitteistokuvauksesta lohkon parametrit, joille asetettiin parametrimuuttujat, sekä sisään- ja ulostuloportit, joihin tulisi myöhemmin liittää väylien signaalit. Lopuksi testattiin, että tähän saakka luodut lähdekoodit kääntyivät.

```
1  module dut_wrapper();
2    import dut_parameters::*;
3
4    // DUT instantiation
5    ip_block #(
6
7        // ip_block parameters
8        .PARAMETER_1(PARAMETER_1),
9        .PARAMETER_2(PARAMETER_2),
10       .PARAMETER_3(PARAMETER_3),
11       .PARAMETER_4(PARAMETER_4),
12       .PARAMETER_5(PARAMETER_5),
13       .PARAMETER_6(PARAMETER_6),
14       .PARAMETER_7(PARAMETER_7),
15       .PARAMETER_8(PARAMETER_8))
16   dut(
17       // Outside signals
18       .clk(),
19       .rst_n(),
20
21       // AHB-lite protcol interface
22       .HSEL(),
23       .HADDR(),
24       .HWRITE(),
25       .HSIZE(),
26       .HTRANS(),
27       .HREADY(),
28       .HWDATA(),
29       .HRESP(),
30       .HREADYOUT(),
31       .HRDATA(),
32
33       // Control signals
34       .CTRL_SIGNAL_1(),
35       .CTRL_SIGNAL_2(),
36       .CTRL_SIGNAL_3(),
37       .CTRL_SIGNAL_4(),
38       .CTRL_SIGNAL_5(),
39       .CTRL_SIGNAL_6());
40 endmodule : dut_wrapper
41
```

KUVA 19. Alustava DUT wrapper

5.4.2 Väylät

DUT wrapperin kuvauksesta puuttui väylien signaalit, joita siirryttiin luomaan DUT:lle. Signaalit voitiin ryhmittää kolmeen eri väylään:

- kontrollisignaalien väylä
- ulkoisten signaalien väylä
- AHB-Lite protokollasignaalien väylä.

Testipenkin toteutuksessa käytettävä AHB-Lite VIP sisälsi valmiin protokollasignaalien väylän, joten suunniteltavaksi jäivät kontrollisignaalien sekä ulkoisten signaalien väylät. Ulkoisina signaaleina olivat kello- ja reset-signaali, jotka oikeassakin toteutuksessa tulisivat ulkoisesti.

Ulkoisille signaaleille luotiin väylä (kuva 20), joka sisälsi kello- ja reset-signaalin sekä kellon generoimisen *always*-proseduurilla. Kellon jaksonaika tuotiin parametrimuuttujien paketista, jota käytettiin viiveen kanssa kellosignaalin generoimisessa ennen kuin invertoitiin kellosignaalin arvo. Viiveen arvoksi asetettiin kellon jaksonaika jaettuna kahdella, joka vastaa yhden ylä- tai alatilan kestoa.

```
1  interface outside_if;
2    import dut_parameters::CLK_CYCLE;
3
4    // Clock and reset
5    logic reset = 1;
6    logic clk = 1;
7
8    // Clock generation
9    always begin
10       #(CLK_CYCLE/2);  // Wait, high state + low state = 2
11       clk = ~clk;
12   end
13 endinterface : outside_if
14
```

KUVA 20. Ulkoisten signaalien väylä

Kontrollisignaalien väylään (kuva 21) luotiin signaalit jokaista DUT:n porttia varten, jotka eivät olleet AHB-Lite protokollan, kellon tai resetin portteja. Signaalit luotiin *logic*-tyyppisiksi, jotta mahdolliset X- ja Z-arvot voitaisiin myös huomata.

```
1 interface control_if;
2   logic ctrl_signal_1;  // output
3   logic ctrl_signal_2;  // input
4   logic ctrl_signal_3;  // input
5   logic ctrl_signal_4;  // output
6   logic ctrl_signal_5;  // output
7   logic ctrl_signal_6;  // output
8 endinterface : control_if
9
```

KUVA 21. Kontrollisignaalien väylä

5.4.3 Väylien liittäminen

DUT wrapper -lohkolle määritettiin argumenteiksi väylien ilmentymät jokaisesta tarvittavasta väylästä, joiden signaalit liitettiin DUT:n portteihin ja VIPin protokollaväylän signaaleihin (kuva 22). Protokollaväylälle oli asetettava testipenkissä käytettävä kello ja reset sekä luotava AHB-Lite protokollan (AMBA 3 AHB-Lite Protocol Specification 2006) dekoodauslogiikka. AHB-Lite protokollan dekoodauslogiikka varten luotiin DUT wrapper -lohkoon IP-lohkon sisääntulosignaali *hsel*, signaali *hready* sekä IP-lohkon protokollarajapinnan ulostuloporttien signaalit, jotka liitettiin DUT:n ilmentymän portteihin. Lopuksi testattiin vielä, että tähän saakka luodut lähdekoodit kääntyivät.

```
1  module dut_wrapper(outside_if i_outside_if, ahb_lite_interface i_ahb_lite_if, control_if i_control_if);
2    import dut_parameters::*;
3
4    // Master to slave signals
5    logic ahb_hsel;
6
7    // Slave to master signals
8    logic ip_block_hready;
9    logic ip_block_hreadyout;
10   logic ip_block_hresp;
11
12   // Assign signals
13   assign i_ahb_lite_if.hclk    = i_outside_if.clk;
14   assign i_ahb_lite_if.hresetn = i_outside_if.reset;
15
16   // AHB Lite decoder and MUX
17   //
18   // ... decoder and MUX logic here ...
19   //
20
21   // DUT instantiation
22   ip_block #(
23
24       // ip_block parameters
25       .PARAMETER_1(PARAMETER_1),
26       .PARAMETER_2(PARAMETER_2),
27       .PARAMETER_3(PARAMETER_3),
28       .PARAMETER_4(PARAMETER_4),
29       .PARAMETER_5(PARAMETER_5),
30       .PARAMETER_6(PARAMETER_6),
31       .PARAMETER_7(PARAMETER_7),
32       .PARAMETER_8(PARAMETER_8))
33   dut(
34       // Outside signals
35       .clk(i_outside_if.clk),
36       .rst_n(i_outside_if.reset),
37
38       // AHB-lite protcol interface
39       .HSEL(ahb_hsel),
40       .HADDR(i_ahb_lite_if.haddr),
41       .HWRITE(i_ahb_lite_if.hwrite),
42       .HSIZE(i_ahb_lite_if.hsize),
43       .HTRANS(i_ahb_lite_if.htrans),
44       .HREADY(i_ahb_lite_if.hready),
45       .HWDATA(i_ahb_lite_if.hwdata),
46       .HRESP(ip_block_hresp),
47       .HREADYOUT(ip_block_hreadyout),
48       .HRDATA(i_ahb_lite_if.hrdata),
49
50       // Control signals
51       .CTRL_SIGNAL_1(i_control_if.ctrl_signal_1),
52       .CTRL_SIGNAL_2(i_control_if.ctrl_signal_2),
53       .CTRL_SIGNAL_3(i_control_if.ctrl_signal_3),
54       .CTRL_SIGNAL_4(i_control_if.ctrl_signal_4),
55       .CTRL_SIGNAL_5(i_control_if.ctrl_signal_5),
56       .CTRL_SIGNAL_6(i_control_if.ctrl_signal_6));
57 endmodule : dut_wrapper
58
```

KUVA 22. Valmis DUT wrapper -lohko

5.4.4 Testipenkkilohko

Testipenkkilohkoon lisätiin *run_test*-metodin kutsun lisäksi väylien ilmentymien luominen sekä DUT wrapperin ilmentymän luominen, jolle annettiin väylien ilmentymät. Testipenkkilohko (kuva 23) viimeisteltiin asettamalla väylien ilmentymät *uvm_config_db*-tietokantaan.

```
1  module testbench_top;
2    `include "uvm_macros.svh"
3    import uvm_pkg::*;
4    import test_pkg::*;
5
6    // Create interfaces
7    outside_if i_outside_if();
8    ahb_lite_interface i_ahb_lite_if();
9    control_if i_control_if();
10
11   // instantiate dut wrapper
12   dut_wrapper dut_wrapper_0(
13       .i_outside_if(i_outside_if),
14       .i_ahb_lite_if(i_ahb_lite_if),
15       .i_control_if(i_control_if));
16
17   initial begin
18     // Add virtual interfaces to the UVM configuration database
19     // Share AHB Lite protocol interface
20     uvm_config_db#(virtual ahb_lite_interface)::set(null, "", "i_ahb_lite_vip.ahb_if", i_ahb_lite_if);
21     // Share control signals' interface
22     uvm_config_db#(virtual control_if)::set(null, "", "i_control_if", i_control_if);
23     // Share the DUT reset and clock
24     uvm_config_db#(virtual outside_if)::set(null, "", "i_outside_if", i_outside_if);
25     run_test();  // Start UVM
26   end
27 endmodule : testbench_top
28
```

KUVA 23. Valmis testipenkkilohko

Testipenkkilohkon jälkeen testattiin, että tähän saakka luodut lähdekoodit kääntyivät ja alettiin luomaan luokkia. Lähdekoodien kääntämiseksi, tiettyjen luokkien olisi oltava luotuna ennen toisia niiden välisten riippuvuuksien takia. Tästä syystä päätettiin aloittaa sekvenssi alkio -luokasta, jota tarvittaisiin sekvenssi-, monitori-, sekvensoija- sekä ajuriluokissa. Luokkien lähdekoodien virheettömyys verifioitiin aina, kun yksi luokka saatiin valmiiksi kääntämällä kaikki siihen saakka luodut lähdekoodit ja korjattiin käännöksessä tulleita virheitä.

5.5 Sekvenssit

Sekvenssien muodostamiseksi tarvitsi luoda sekvenssi- sekä sekvenssi alkio -luokat. Sekvenssiluokka toteutettiin siten, että sekvenssejä pystyttiin luomaan kontrolloidusti testiluokasta satunnaistamisen sijan.

Sekvenssi alkio -luokkaan luotiin ainoastaan väylän sisään- ja ulostulosignaalien arvojen siirtämiseen tarvittavat muuttujat. Kyseistä luokkaa käytettiin ajuri- sekä monitoriluokassa väylän arvojen kanssa kommunikoimiseksi.

5.5.1 Sekvenssi alkio -luokka

Sekvensseihin liittyville luokille eli sekvenssi- ja sekvenssi alkio -luokille luotiin paketti (kuva 24) kuten testiluokille tehtiin alustavassa testipenkissä. Tyhjät luokkien lähdekooditiedostot voitiin myös jo luoda ja sisällyttää pakettiin valmiiksi. Pakettiin oli muistettava järjestää *`include*-kääntäjädirektiiveillä sisällytettävät luokkien lähdekooditiedostot oikein kääntämisjärjestyksen takia.

```
package seq_pkg;
  `include "uvm_macros.svh"
  import uvm_pkg::*;

  // Include project-specific sequence classes
  `include "seq_item.svh"
  `include "seq.svh"
endpackage : seq_pkg
```

KUVA 24. Sekvensseihin liittyvien lähdekoodien paketti

Sekvenssi alkio -luokkaan (kuva 25) ei ollut paljon luotavaa. Luokka rekisteröitiin tehtaalle *`uvm_object_utils_begin*- ja *`uvm_object_utils_end*-makroilla, jotta voitiin käyttää *`uvm_field_** -makroja ja käyttää *print*-metodia. Tehtaalle rekisteröinnin lisäksi luokalle luotiin jäsenmuuttujat, jotka vastasivat kontrollisignaaleja ja lopuksi luotiin luokan rakentajafunktio.

```
`ifndef SEQ_ITEM_SVH_
`define SEQ_ITEM_SVH_

class seq_item extends uvm_sequence_item;
  // Class members
  rand logic m_ctrl_signal_2; // Input
  rand logic m_ctrl_signal_3; // Input
  logic      m_ctrl_signal_1; // Output
  logic      m_ctrl_signal_4; // Output
  logic      m_ctrl_signal_5; // Output
  logic      m_ctrl_signal_6; // Output

  // Register the class to the UVM factory
  // Register class members to utilize the print-method
  `uvm_object_utils_begin(seq_item)
    `uvm_field_int(m_ctrl_signal_2, UVM_ALL_ON | UVM_DEC)
    `uvm_field_int(m_ctrl_signal_3, UVM_ALL_ON | UVM_DEC)
    `uvm_field_int(m_ctrl_signal_1, UVM_ALL_ON | UVM_DEC)
    `uvm_field_int(m_ctrl_signal_4, UVM_ALL_ON | UVM_DEC)
    `uvm_field_int(m_ctrl_signal_5, UVM_ALL_ON | UVM_DEC)
    `uvm_field_int(m_ctrl_signal_6, UVM_ALL_ON | UVM_DEC)
  `uvm_object_utils_end

  // Method prototypes
  extern function new(string name = "seq_item");  // Class constructor
endclass : seq_item

// Function: new
function seq_item::new(string name = "seq_item");
  super.new(name);
endfunction : new

`endif  // SEQ_ITEM_SVH_
```

KUVA 25. Sekvenssi alkio -luokka

5.5.2 Sekvenssiluokka

Sekvenssiluokka parametrisoitiin edellä luodulla sekvenssi alkio -luokalla, rekisteröitiin *`uvm_object_utils*-makrolla tehtaalle ja määritettiin kontrollisignaalien väylää vastaaville sisääntulosignaaleille jäsenmuuttujat. Tarkoituksena oli tehdä sekvensseistä kontrolloituja eli satunnaistamista tai silmukkaa ei toteutettu *body*-metodiin.

Luokalle (kuva 26) luotiin rakentajafunktio ja *body*-metodi, joka suunniteltiin käytettäväksi *get_next_item*-metodia käyttävän ajurin kanssa. Metodissa *body* luotiin uusi sekvenssi alkio, kutsuttiin *start_item*-metodia ja asetettiin sekvenssiluokan jäsenmuuttujien arvot sekvenssi alkion jäsenmuuttujien arvoiksi. Metodin lopuksi kutsuttiin *finish_item*-metodia ilmoittamaan, että sekvenssi on loppunut.

```
`ifndef SEQ_SVH_
`define SEQ_SVH_

class seq extends uvm_sequence #(seq_item);
  `uvm_object_utils(seq)  // Register the class to the UVM factory

  // Class members
  rand logic    m_ctrl_signal_2;  // input signal
  rand logic    m_ctrl_signal_3;   // input signal
  seq_item m_seq_item_h;

  // Method prototypes
  extern function
      new(string name = "seq");  // Class constructor
  extern virtual task
      body();                    // Construct sequence and start communication
endclass : seq

// Function: new
function seq::new(string name = "seq");
  super.new(name);
endfunction : new

// Virtual task: body
task seq::body();
  // Create sequence item object
  m_seq_item_h = seq_item::type_id::create("m_seq_item_h");

  // Send sequence item object to the sequencer
  start_item(m_seq_item_h);
  // Set values to sequence item object's class members
  m_seq_item_h.m_ctrl_signal_2 = m_ctrl_signal_2;
  m_seq_item_h.m_ctrl_signal_3 = m_ctrl_signal_3;
  // Finish item and wait for the driver's response from sequencer
  finish_item(m_seq_item_h);
endtask : body

`endif  // SEQ_SVH_
```

KUVA 26. Sekvenssiluokka

5.6 Sekvenssien ajaminen

Sekvenssien ajamiseksi tarvittiin ajuri sekä sekvensoija, joka välitti sekvenssiluokalta sekvenssejä ajurille. Näitä UVM-komponentteja varten täytyi olla valmiina sekvenssi alkio -luokka, jolla niiden luokat parametrisoitiin.

Sekvensoijan ja ajurin lisäksi kehitettiin tässä vaiheessa alustava agentti- ja ympäristöluokka. Lopuksi varmistettiin ajurin toiminta muuttamalla sekvenssi alkion muuttujien arvoja ja kutsumalla sekvenssiluokan *start*-metodia.

5.6.1 Sekvensoijaluokka

Ympäristöön liittyville luokille eli sekvensoija-, ajuri-, monitori-, agentti-, rekisterirajapinnan ympäristö - ja ympäristöluokille luotiin paketti kuten testiluokille ja sekvensseihin liittyville luokille tehtiin. Tyhjät luokkien lähdekooditiedostot voitiin myös jo luoda ja sisällyttää pakettiin valmiiksi. Pakettiin (kuva 27) oli muistettava järjestää *`include*-kääntäjädirektiiveillä sisällytettävät luokkien lähdekooditiedostot oikein kääntämisjärjestyksen takia ja tuoda sekvensseihin liittyvien luokkien paketti.

```
package env_pkg;
  `include "uvm_macros.svh"
  import uvm_pkg::*;

  // Import other packages
  import seq_pkg::*;

  // Include project-specific class headers
  `include "sequencer.svh"
  `include "driver.svh"
  `include "monitor.svh"
  `include "agent.svh"
  `include "env.svh"
endpackage : env_pkg
```

KUVA 27. Ympäristöön liittyvien luokkien paketti

Testipenkille luotiin sekvensoijaluokka, jotta tuotaisiin paremmin esille sen olemassaolo, vaikka sen kantaluokasta olisi voitu luoda suoraan olio ympäristöön. Sekvensoijaluokka periytettiin *uvm_sequencer*-luokasta, rekisteröitiin tehtaalle ja luokalle luotiin rakentajafunktio. Kuvassa 28 esitetään sekvensoijaluokka.

```
`ifndef SEQUENCER_SVH_
`define SEQUENCER_SVH_

// Class uvm_sequencer #(type REQ = uvm_sequence_item, RSP = REQ) extends uvm_sequencer_param_base #(REQ, RSP);
// RSP is left to equal REQ
class sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(sequencer)  // Register the class to the UVM component factory

  // Method prototypes
  extern function
      new(string name, uvm_component parent);  // Class constructor
endclass : sequencer

// Function: new
function sequencer::new(string name, uvm_component parent);
  super.new(name, parent);
endfunction : new

`endif  // SEQUENCER_SVH_
```

KUVA 28. Sekvensoijaluokka

5.6.2 Ajuriluokka

Ajuriluokka (kuva 29) parametrisoitiin samalla sekvenssi alkio -luokalla kuin sekvenssiluokkakin. Luokalle luotiin jäsenmuuttujiksi virtuaalinen kontrollisignaalien väylän ilmentymä sekvenssien ajamiseksi DUT:lle sekä sekvenssi alkio -luokka tyyppinen muuttuja vastaanottamaan ajettavien olioiden dataa sekvensoijalta.

```
1  `ifndef DRIVER_SVH_
2  `define DRIVER_SVH_
3
4  class driver extends uvm_driver #(seq_item);
5    `uvm_component_utils(driver)  // Register the class to the factory
6
7    // Class members
8    virtual control_if m_i_control_vif;  // control interface
9    seq_item           m_seq_item_h;     // sequence item object handle
10
11   // Method prototypes
12   extern function
13       new(string name, uvm_component parent);  // Class constructor
14   extern virtual function void
15       build_phase(uvm_phase phase);            // Used to build objects and connect interfaces
16   extern virtual task
17       run_phase(uvm_phase phase);              // Used to drive values to the DUT
18 endclass : driver
19
```

KUVA 29. Ajuri

Luokkaan luotiin rakentaja-, *build_phase*- ja *run_phase*-metodit. Metodissa *build_phase* liitettiin virtuaalinen kontrollisignaalien väylä DUT wrapperissa vastaavaan väylään käyttämällä *uvm_config_db*-tietokannan *get*-metodia ja luotiin sekvenssi alkio -luokan olio. Metodissa *run_phase* puolestaan luotiin käskyt ajamaan uusia arvoja väylälle.

Metodiin *run_phase* luotiin uusien arvojen ajamiseksi ikuinen silmukka, jonka sisällä kutsuttiin metodia *get_next_item* hakemaan uusi sekvenssi alkio, asetettiin sekvenssi alkion jäsenmuuttujien arvot virtuaalisen väylän signaalien arvoiksi ja kutsuttiin lopuksi *item_done*-metodia. Metodille *run_phase* piti muistaa olla asettamatta *phase.raise_objection*-metodin kutsua sen sisältämän ikuisen silmukan takia. Kuvassa 30 esitetään ajuriluokan metodien lähdekoodit.

```
20    // Function: new
21    function driver::new(string name, uvm_component parent);
22      super.new(name,parent);
23    endfunction : new
24
25    // Virtual function void: connect_phase
26    function void driver::build_phase(uvm_phase phase);
27      super.build_phase(phase);
28
29      // Connects driver's virtual interface to the DUT's interface or stops simulation if not possible
30      if (!uvm_config_db #(virtual control_if)::get(this, "", "i_control_if", m_i_control_vif)) begin
31        `uvm_fatal(get_full_name (), "Can't get control_if from config_db")
32      end
33
34      // Create sequence item object and assign the handle to the class member
35      m_seq_item_h = seq_item::type_id::create("m_seq_item_h");
36    endfunction : build_phase
37
38    // Virtual task: run_phase
39    task driver::run_phase(uvm_phase phase);
40      super.run_phase(phase);
41
42      forever begin : dut_stimulus
43        // Get next sequence_item object from the sequencer via TLM-port using blocking read
44        seq_item_port.get_next_item(m_seq_item_h);
45        // Drive the data to the individual DUT ports from the transaction using a virtual interface
46        m_i_control_vif.ctrl_signal_2 <= m_seq_item_h.m_ctrl_signal_2;
47        m_i_control_vif.ctrl_signal_3 <= m_seq_item_h.m_ctrl_signal_3;
48        // Tell the sequencer the transaction is done
49        seq_item_port.item_done(m_seq_item_h);
50      end : dut_stimulus
51    endtask : run_phase
52
53    `endif  // DRIVER_SVH_
```

KUVA 30. Ajuriluokan metodit

5.6.3 Ajurin testaaminen

Tarkoituksena oli testata ajurin ja sekvensoijan luomisen jälkeen, että sekvenssien tai arvojen syöttäminen DUT:lle onnistuu. Testin tiedettiin onnistuneen, jos nähtiin muutoksia signaaleissa simuloidessa graafisella käyttöliittymällä. Ajurin testaamiseksi luotiin alustava agentti ja ympäristö.

Ajuria olisi voinut testata myös luomalla suoraan testiluokkaan ajuriluokasta olio ja tuottaa sekvenssejä, mutta agentti ja ympäristö olisi kuitenkin luotava jossain vaiheessa. Agentissa tehtävä ajurin liittäminen sekvensoijaan olisi lisäksi jouduttu muodostamaan testiluokassa ja poistamaan myöhemmin. Tästä johtuen oli loogisempaa testata ajuria luomalla alustava agentti ja ympäristö, jotka sisältäisivät pelkästään ajurin ja sekvensoijan.

Agenttiluokkaan (kuva 31) luotiin jäsenmuuttujiksi ajuri- ja sekvensoijaluokka tyyppiset muuttujat sekä luotiin rakentaja-, *build_phase*- ja *connect_phase*-metodi. Metodissa *build_phase* luotiin ajuri- ja sekvensoijaluokka oliot, jonka jälkeen lähdekoodit yritettiin kääntää, mutta kääntäjältä tuli virheilmoitus fatal error ”bad handle or reference”. Virheilmoitukset tulivat, koska agenttiluokan metodissa *con-*

nect_phase ei ollut vielä liitettynä ajurin ja sekvensoijan TLM-porttityyppejä *connect*-metodilla. Metodissa *connect_phase* ajurin ja sekvensoijan liittämisen lisäksi asetettiin sekvensoijan olion kahva *uvm_config_db*-tietokantaan, jonka jälkeen agentti oli valmiina ajurin testaamista varten.

```
`ifndef AGENT_SVH_
`define AGENT_SVH_

class agent extends uvm_agent;
  `uvm_component_utils(agent)  // Register the class to the factory

  // Class members
  driver    m_driver_h;     // driver
  sequencer m_sequencer_h;  // sequencer

  // Method prototypes
  extern function
      new(string name, uvm_component parent);  // Class constructor
  extern virtual function void
      build_phase(uvm_phase phase);            // Used to build components
  extern virtual function void
      connect_phase(uvm_phase phase);          // Used to make connections between components
endclass : agent

// Function: new
function agent::new(string name, uvm_component parent);
  super.new(name,parent);
endfunction : new

// Virtual Function void: build_phase
function void agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Create UVM components
    m_driver_h    = driver::type_id::create("m_driver_h", this);
    m_sequencer_h = sequencer::type_id::create("m_sequencer_h", this);
    // Set the agent's sequencer object to config_db
    uvm_config_db #(sequencer)::set(null, "", "m_sequencer_h", m_sequencer_h);
endfunction : build_phase

// Virtual Function void: connect_phase
function void agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

    // Connect the driver's TLM port to the sequencer's TLM export
    m_driver_h.seq_item_port.connect(m_sequencer_h.seq_item_export);
endfunction : connect_phase

`endif  // AGENT_SVH_
```

KUVA 31. Alustava agenttiluokka ajurin testaamiseksi

Ympäristöluokkaan luotiin jäsenmuuttujaksi agenttiluokka tyyppinen muuttuja ja luotiin metodi *build_phase*, jossa luotiin agenttiluokasta olio. Alustavan ympäristön (kuva 32) jälkeen voitiin kirjoittaa testiluokassa muutamia syötteitä käyttäen sekvenssiluokan *start*-metodia.

```
1   `ifndef ENV_SVH_
2   `define ENV_SVH_
3
4   class env extends uvm_env;
5     `uvm_component_utils(env)
6
7     // Class members
8     agent m_agent_h;  // agent
9
10    // Method prototypes
11    extern function
12        new(string name = "env", uvm_component parent);  // Class constructor
13    extern virtual function void
14        build_phase(uvm_phase phase);                     // Used to build components
15  endclass : env
16
17  // Function: new
18  function env::new(string name = "env", uvm_component parent);
19    super.new(name, parent);
20  endfunction : new
21
22  // Virtual Function: build_phase
23  function void env::build_phase(uvm_phase phase);
24    super.build_phase(phase);
25
26    // Create objects via factory and store objects into handles
27    m_agent_h = agent::type_id::create("m_agent_h", this);
28  endfunction : build_phase
29
30  `endif  // ENV_SVH_
31
```

KUVA 32. Alustava ympäristöluokka ajurin testaamiseksi

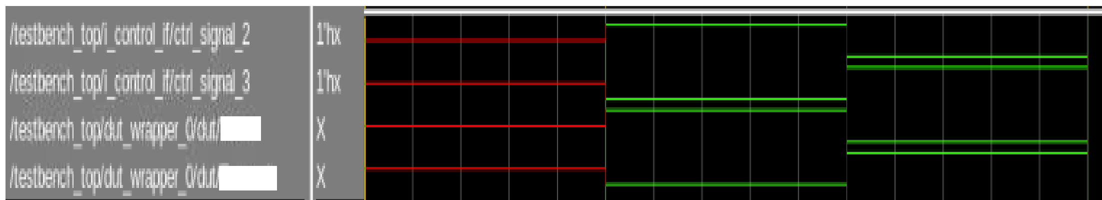
Alustavaan testiluokkaan lisättiin ympäristö-, sekvenssi- ja sekvensoijaluokka tyyppiset muuttujat käyttääkseen *start*-metodia. Testiluokalle luotiin myös *build_phase*-metodi, jossa luotiin niille oliot tehtaan avulla. Luokkien käyttämiseksi piti tuoda ensin ympäristö sekä sekvensseihin liittyvien luokkien paketit testiluokkien paketissa.

Testiluokkaan luotiin agentissa *uvm_config_db*-tietokantaan asetetun sekvensoijan olion hakemiseksi *start_of_simulation_phase*-metodi. Metodia voitiin jatkossa käyttää hakemaan testejä varten tarvittavat resurssit *uvm_config_db*-tietokannasta. Testiluokan (kuva 33) *run_phase*-metodissa kirjoitettiin lopuksi muutamat syötteet *start*-metodilla, joilla testattiin, että yhteydet toimivat ja DUT:n porteissa näkyi muutoksia.

```
`ifndef TEST_SVH_
`define TEST_SVH_

class test extends uvm_test;
  // Register the class to the UVM component factory
  `uvm_component_utils(test)
  `timescale 1ns/1ps

  // Class members
  env       m_env_h;        // environment
  seq       m_seq_h;        // sequence
  sequencer m_sequencer_h;  // sequencer

  // Method prototypes
  extern function
      new(string name = "test", uvm_component parent);  // Class constructor
  extern virtual function void
      build_phase(uvm_phase phase);                      // Create component objects
  extern virtual task
      run_phase(uvm_phase phase);                        // Run phase
  extern virtual function void
      start_of_simulation_phase(uvm_phase phase);        // Get items from uvm_config_db
endclass : test

// Function: new
//
// - Creates test objects
function test::new(string name = "test", uvm_component parent);
  super.new(name,parent);
endfunction : new

// virtual function void: build_phase
function void test::build_phase(uvm_phase phase);

  // create objects
  m_env_h = env::type_id::create("m_env_h", this);
  m_seq_h = seq::type_id::create("m_seq_h", this);
endfunction : build_phase

// Virtual task: run_phase
//
// TODO this is used to construct a working TB
// - Prints Hello World!
task test::run_phase(uvm_phase phase);
  phase.raise_objection(this);

  #10
  m_seq_h.m_ctrl_signal_2 = 1;
  m_seq_h.m_ctrl_signal_3 = 0;
  m_seq_h.start(m_sequencer_h);
  #100
  m_seq_h.m_ctrl_signal_2 = 0;
  m_seq_h.m_ctrl_signal_3 = 1;
  m_seq_h.start(m_sequencer_h);
  #100
  `uvm_info ("test::run_phase","Hello World!",UVM_LOW);

  phase.drop_objection(this);
endtask : run_phase

// Function void: start_of_simulation_phase
function void test::start_of_simulation_phase(uvm_phase phase);

  // Get the sequencer object from config_db
  if (!uvm_config_db#(sequencer)::get(this, "", "m_sequencer_h", m_sequencer_h)) begin
    `uvm_fatal("test::start_of_simulation_phase", "Cannot get m_sequencer_h from config_db");
  end
endfunction : start_of_simulation_phase

`endif  // TEST_SVH_
```

KUVA 33. Sekvenssien kirjoittaminen testiluokasta

Ajurin toiminta testattiin invertoimalla DUT:n sisääntulosignaalien arvot ja laittamalla viivettä *start*-metodin kutsujen väliin. Lopuksi käännettiin lähdekoodit ja simuloitiin DUT:n käyttäytymistä. Kuvassa 34 esitetään aaltomuotoeditorissa signaalien käyttäytyminen, jolla varmistettiin ajurin toimivan oikein.



KUVA 34. Ajurin toiminnan varmistaminen aaltomuotoeditorilla

5.7 Sekvenssien monitorointi

DUT:n ulostulosignaalien monitoroimiseksi tarvitsi luoda monitoriluokka. Monitoriluokan lisäksi täytyi lisätä monitori agenttiluokkaan ja liittää monitorin ulostulo agentin ulostuloon.

Lopuksi testattiin vielä monitorin toiminta aikaisemmin ajurille luoduilla syötteillä. Monitoriluokassa kutsuttiin testausta varten tulostuksia, jotta nähtiin signaalien tilanmuutokset.

5.7.1 Monitoriluokka

Ajurin toiminnan varmistamisen jälkeen siirryttiin kehittämään monitoriluokkaa, joka oli ainoa puuttuva komponentti kontrollisignaalien väylän agentissa. Monitoriluokkaan luotiin jäsenmuuttujiksi virtuaaliset kontrollisignaalien ja ulkoisten signaalien väylien ilmentymät sekä *uvm_analysis_port* ja sekvenssi alkio -luokka tyyppiset muuttujat, jotta kontrolliväylän signaalien arvot voitiin siirtää tulostaululle.

Luokalle luotiin rakentajafunktio, *build_phase*-, *run_phase*- ja *start_of_simulation_phase*-metodi. Metodissa *build_phase* luotiin TLM-portti *new*-metodilla, sekvenssi alkio -luokan olio *create*-metodilla sekä haettiin *get*-metodilla *uvm_config_db*-tietokannasta väylien ilmentymät. Metodissa *start_of_simulation_phase* asetettiin kontrolliväylän signaalien arvot sekvenssi alkion jäsenmuuttujien arvoiksi ennen simulaation aloittamista. Lopuksi luotiin *run_phase*-metodissa logiikka väylän signaalien arvojen siirtämiseksi tulostaululle.

Metodin *run_phase* alussa kutsuttiin *write*-metodia, jolla sekvenssi alkiot siirrettiin TLM-portin kautta tulostaululle. Uusien arvojen siirtämiseksi simulaation aikana luotiin *fork-join*-lohko sekä *forever*-silmukka, jonka sisällä annettiin tapahtumakontrollilauseke. Tapahtumakontrollilauseke tarkkaili ulkoisten signaalien väylän kellon nousureunaa, jolloin siirrettiin uudet arvot väylältä asettamalla sen signaalien arvot sekvenssi alkion jäsenmuuttujien arvoiksi. Monitoriluokan (kuva 35) *run_phase*-metodiin lisättiin myös tulostuksia väylän signaalien arvoista, jotta voitiin simuloidessa seurata, mitä tapahtuu.

```
1   `ifndef MONITOR_SVH_
2   `define MONITOR_SVH_
3
4   class monitor extends uvm_monitor;
5     `timescale 1ns/1ps
6     `uvm_component_utils(monitor)  // Register the class to the factory
7
8     // Class members
9     virtual control_if            m_i_control_vif;       // receiver interface
10    virtual outside_if            m_i_outside_vif;       // testbench interface
11    seq_item                      m_seq_item_h;          // sequence item object handle
12    uvm_analysis_port #(seq_item) m_mon_analysis_port;   // TLM-port to agent
13
14    // Method prototypes
15    extern function
16        new(string name, uvm_component parent);       // Class constructor
17    extern virtual function void
18        build_phase(uvm_phase phase);                  // Used to build needed components and ports
19    extern virtual function void
20        start_of_simulation_phase(uvm_phase phase);   // Used to set inital values to signals
21    extern virtual task
22        run_phase(uvm_phase phase);                    // Used to detect signal changes
23  endclass : monitor
24
25  // Function: new
26  function monitor::new(string name, uvm_component parent);
27    super.new(name,parent);
28  endfunction : new
29
30  // Virtual function void: build_phase
31  function void monitor::build_phase(uvm_phase phase);
32    super.build_phase(phase);
33
34    // Create sequence item object
35    m_seq_item_h = seq_item::type_id::create("m_seq_item_h", this);
36
37    // Create instance of the declared analysis port
38    m_mon_analysis_port = new("m_mon_analysis_port", this);
39
40    // Connect virtual interfaces
41    if (!uvm_config_db #(virtual control_if)::get(this, "", "i_control_if", m_i_control_vif)) begin
42      `uvm_fatal(get_full_name (), "Can't get i_control_if from config_db");
43    end
44    if (!uvm_config_db #(virtual outside_if)::get(this, "", "i_outside_if", m_i_outside_vif)) begin
45      `uvm_fatal(get_full_name (), "Can't get i_outside_if from config_db");
46    end
47  endfunction : build_phase
48
49  // Virtual function void: start_of_simulation_phase
50  function void monitor::start_of_simulation_phase(uvm_phase phase);
51    m_seq_item_h.m_ctrl_signal_1 = m_i_control_vif.ctrl_signal_1;
52    m_seq_item_h.m_ctrl_signal_2 = m_i_control_vif.ctrl_signal_2;
53    m_seq_item_h.m_ctrl_signal_3 = m_i_control_vif.ctrl_signal_3;
54    m_seq_item_h.m_ctrl_signal_4 = m_i_control_vif.ctrl_signal_4;
55    m_seq_item_h.m_ctrl_signal_5 = m_i_control_vif.ctrl_signal_5;
56    m_seq_item_h.m_ctrl_signal_6 = m_i_control_vif.ctrl_signal_6;
57  endfunction : start_of_simulation_phase
58
59  // Virtual task: run_phase
60  task monitor::run_phase(uvm_phase phase);
61    forever begin : monitor_signals
62      // Broadcast the object to the scoreboard
63      m_mon_analysis_port.write(m_seq_item_h);
64      fork
65        // Transfer new interface values at every positive edge of the clock
66        @ (posedge m_i_outside_vif.clk) begin
67          m_seq_item_h.m_ctrl_signal_1 = m_i_control_vif.ctrl_signal_1;
68          m_seq_item_h.m_ctrl_signal_2 = m_i_control_vif.ctrl_signal_2;
69          m_seq_item_h.m_ctrl_signal_3 = m_i_control_vif.ctrl_signal_3;
70          m_seq_item_h.m_ctrl_signal_4 = m_i_control_vif.ctrl_signal_4;
71          m_seq_item_h.m_ctrl_signal_5 = m_i_control_vif.ctrl_signal_5;
72          m_seq_item_h.m_ctrl_signal_6 = m_i_control_vif.ctrl_signal_6;
73          `uvm_info("monitor::run_phase",$sformatf("\ncontrol signal 1: %0d \
74                                                    \ncontrol signal 2: %0d \
75                                                    \ncontrol signal 3: %0d \
76                                                    \ncontrol signal 4: %0d \
77                                                    \ncontrol signal 5: %0d \
78                                                    \ncontrol signal 6: %0d",
79                                                    m_seq_item_h.m_ctrl_signal_1,
80                                                    m_seq_item_h.m_ctrl_signal_2,
81                                                    m_seq_item_h.m_ctrl_signal_3,
82                                                    m_seq_item_h.m_ctrl_signal_4,
83                                                    m_seq_item_h.m_ctrl_signal_5,
84                                                    m_seq_item_h.m_ctrl_signal_6), UVM_LOW)
85        end
86      join
87    end : monitor_signals
88  endtask : run_phase
89
90  `endif  // MONITOR_SVH_
91
```

KUVA 35. Monitoriluokka

5.7.2 Monitorin testaaminen

Monitorin testaamiseksi lisättiin agenttiluokkaan jäsenmuuttujiksi monitoriluokka ja *uvm_analysis_port* tyyppiset muuttujat sekä luotiin niille oliot *build_phase*-metodissa. Agenttiluokan metodissa *connect_phase* liitettiin monitorin TLM-portti ja juuri luotu agentin TLM-portti *connect*-metodilla (kuva 36).

```
`ifndef AGENT_SVH_
`define AGENT_SVH_

class agent extends uvm_agent;
  `uvm_component_utils(agent)  // Register the class to the factory

  // Class members
  driver                        m_driver_h;     // driver
  sequencer                     m_sequencer_h;  // sequencer
  monitor                       m_monitor_h;    // monitor
  uvm_analysis_port #(seq_item) m_agent_analysis_port;

  // Method prototypes
  extern function
      new(string name, uvm_component parent);  // Class constructor
  extern virtual function void
      build_phase(uvm_phase phase);            // Used to build components
  extern virtual function void
      connect_phase(uvm_phase phase);          // Used to make connections between components
endclass : agent

// Function: new
function agent::new(string name, uvm_component parent);
  super.new(name,parent);
endfunction : new

// Virtual Function void: build_phase
function void agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Create UVM components
  m_driver_h    = driver::type_id::create("m_driver_h", this);
  m_sequencer_h = sequencer::type_id::create("m_sequencer_h", this);
  m_monitor_h   = monitor::type_id::create("m_monitor_h", this);

  // Create agent's TLM port
  m_agent_analysis_port = new("m_agent_analysis_port", this);

  // Set the agent's sequencer object to config_db
  uvm_config_db #(sequencer)::set(null, "", "m_sequencer_h", m_sequencer_h);
endfunction : build_phase

// Virtual Function void: connect_phase
function void agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  // Connect the driver's TLM port to the sequencer's TLM export
  m_driver_h.seq_item_port.connect(m_sequencer_h.seq_item_export);
  // Connect the monitor's TLM port to agent's TLM port
  m_monitor_h.m_mon_analysis_port.connect(m_agent_analysis_port);
endfunction : connect_phase

`endif  // AGENT_SVH_
```

KUVA 36. Valmis agenttiluokka

Agentti oli nyt valmis ja monitori voitiin testata ajamalla simulaatio, jolla tarkistettiin tähän saakka tehtyjen lähdekoodien virheettömyys. Kuvassa 37 on simulaation tulokset ilman käyttöliittymää.

```
# UVM_INFO /monitor.svh(73) @ 50000: uvm_test_top.m_env_h.m_agent_h.m_mon
itor_h [monitor::run_phase]
# control signal 1: 0
# control signal 2: 1
# control signal 3: 0
# control signal 4: 1
# control signal 5: 1
# control signal 6: 0
# UVM_INFO /monitor.svh(73) @ 100000: uvm_test_top.m_env_h.m_agent_h.m_mo
nitor_h [monitor::run_phase]
# control signal 1: 0
# control signal 2: 1
# control signal 3: 0
# control signal 4: 1
# control signal 5: 0
# control signal 6: 0
# UVM_INFO /monitor.svh(73) @ 150000: uvm_test_top.m_env_h.m_agent_h.m_mo
nitor_h [monitor::run_phase]
# control signal 1: 0
# control signal 2: 0
# control signal 3: 1
# control signal 4: 1
# control signal 5: 0
# control signal 6: 0
# UVM_INFO /monitor.svh(73) @ 200000: uvm_test_top.m_env_h.m_agent_h.m_mo
nitor_h [monitor::run_phase]
# control signal 1: 0
# control signal 2: 0
# control signal 3: 1
# control signal 4: 1
# control signal 5: 0
# control signal 6: 0
# UVM_INFO /test.svh(56) @ 210000: uvm_test_top [test::run_phase] Hel
lo World!
# UVM_INFO /uvm_objection.svh(1270) @ 210000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO /uvm_report_server.svh(847) @ 210000: reporter [UVM/REPORT/SERVER]
# --- UVM Report Summary ---
#
# Quit count :     0 of  1000
# ** Report counts by severity
# UVM_INFO :    9
# UVM_WARNING :    1
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [MAXQUITSET]     1
# [RNTST]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
# [UVM/RSRC/NOREGEX]     1
# [monitor::run_phase]     4
# [test::run_phase]     1
```

KUVA 37. Agentin, monitorin ja ajurin testaaminen

5.8 Rekistereiden lukeminen ja kirjoittaminen

RAL oli automaattisesti generoitu käyttämällä yrityksen sisäistä työkalua ja käytössä ollut AHB-Lite VIP sisälsi protokolla-adapterin ja -agentin. Tästä syystä johtuen, niiden lähdekoodeja ei esitetä.

VIPin UVM-komponentit luotiin RAL-ympäristöön, jossa tehtiin kaikki tarvittavat toimenpiteet. RAL-ympäristön tarkoituksena oli vähentää tarvittavia toimenpiteitä itse testipenkin ympäristössä. Lopuksi tehtiin rekisterikirjoituksia testiluokassa, joiden avulla testattiin RAL-ympäristön toiminta.

5.8.1 RAL-ympäristö

RALin luokille luotiin paketti, jotta niiden lähdekoodeihin olisi helpompi viitata ja ympäristöön liittyvien luokkien pakettiin tehtiin lisäyksiä. Lisäyksenä sisällytettiin RAL-ympäristöluokka \`*include*-kääntäjädirektiivillä ja tuotiin RALin sekä AHB-Lite VIPin pakettien koko sisältö.

Testipenkille muodostettiin RAL-ympäristöluokka, jolla vähennettiin tarvittavia toimenpiteitä itse ympäristössä. RAL-ympäristöluokkaan luotiin jäsenmuuttuja *uvm_analysis_port* AHB-Lite sekvenssi alkioiden siirtämiseksi tulostaululle ja jäsenmuuttujat RALin sekä protokolla-adapterin ja -agentin olioita varten. RAL-ympäristön *build_phase*-metodissa

- luotiin RAL-, AHB-Lite adapteri- ja agenttiluokan oliot
- luotiin TLM-portti kutsumalla sen rakentajafunktiota
- kutsuttiin RALin *build*-, *lock*- ja *reset*-metodeja
- asetettiin RALin olion kahva *uvm_config_db*-tietokantaan
- tehtiin RALin syötteenennustuksesta implisiittinen kutsumalla *set_auto_predict*-metodia ja antamalla sille arvoksi 1.

Metodissa *connect_phase* liitettiin vielä AHB-Lite VIPin agentin TLM-portti RAL-ympäristöluokan TLM-porttiin ja kutsuttiin RALin *set_sequencer*-metodi, jolle asetettiin argumenteiksi AHB-Lite VIPin agentin sekvensoija ja adapteri. Kuvassa 38 on lopullinen RAL-ympäristöluokka.

```
1  `ifndef REG_ENV_SVH_
2  `define REG_ENV_SVH_
3
4  class reg_env extends uvm_env;
5    `uvm_component_param_utils(reg_env)
6
7    // Class members
8    ral                                   m_ral_h;                      // RAL
9    ahb_lite_adapter                      m_ahb_lite_adapter_h;         // Adapter
10   ahb_lite_agent                        m_ahb_lite_agent_h;           // Agent
11   uvm_analysis_port #(ahb_lite_seq_item) m_ahb_lite_analysis_port;    // TLM port to the scoreboard
12
13   extern function
14       new(string name = "reg_env", uvm_component parent);  // Constructor
15   extern virtual function void
16       build_phase(uvm_phase phase);                        // Used to build components
17   extern virtual function void
18       connect_phase(uvm_phase phase);                      // Used to connect components
19 endclass : reg_env
20
21 // Function: new
22 function reg_env::new(string name = "reg_env", uvm_component parent);
23   super.new(name, parent);
24 endfunction : new
25
26 // Virtual function void: build_phase
27 function void reg_env::build_phase(uvm_phase phase);
28   super.build_phase(phase);
29
30   // Create ral, adapter and agent
31   m_ral_h                  = ral::type_id::create("m_ral_h", this);
32   m_ahb_lite_adapter_h     = ahb_lite_adapter::type_id::create("m_ahb_lite_adapter_h",this);
33   m_ahb_lite_agent_h = ahb_lite_agent::type_id::create("m_ahb_lite_agent_h", this);
34
35   // Create TLM-port
36   m_ahb_lite_analysis_port = new("m_ahb_lite_analysis_port", this);
37
38   m_ral_h.build();
39   m_ral_h.lock_model();
40   m_ral_h.reset();
41
42   // Set auto prediction of the register model ON
43   m_ral_h.default_map.set_auto_predict(1);
44
45   // Set the RAL's object handle to the configuration database
46   uvm_config_db #(ral)::set(null, "", "ral", m_ral_h);
47 endfunction : build_phase
48
49 // Virtual function void: connect_phase
50 function void reg_env::connect_phase(uvm_phase phase);
51   super.connect_phase(phase);
52
53   m_ahb_lite_agent_h.ahb_lite_analysis_port.connect(m_ahb_lite_analysis_port);
54   m_ral_h.default_map.set_sequencer(m_ahb_lite_agent_h.sequencer, m_ahb_lite_adapter_h);
55 endfunction : connect_phase
56
57 `endif  // REG_ENV_SVH_
58
```

KUVA 38. Rekisterirajapinnan ympäristöluokka

5.8.2 RAL-ympäristön testaaminen

RAL-ympäristön testaamiseksi lisättiin ympäristöluokkaan jäsenmuuttujaksi RAL-ympäristöluokka tyyppinen muuttuja sekä luotiin sille olio *build_phase*-metodissa. Testiluokkaan luotiin (kuva 39) VIPin konfiguraatioluokka ja RAL tyyppiset muuttujat sekä *uvm_status_e status* muuttuja rekisterikirjoitusten ja -lukujen tekemiseksi. Luokkien käyttämiseksi piti kuitenkin ensin tuoda testiluokkien pakettiin RALin sekä AHB-Lite VIPin koko paketin sisältö *import*-käskyillä.

```
1  `ifndef TEST_SVH_
2  `define TEST_SVH_
3
4  class test extends uvm_test;
5    // Register the class to the UVM component factory
6    `uvm_component_utils(test)
7    `timescale 1ns/1ps
8
9    // Class members
10   env                 m_env_h;        // environment
11   seq                 m_seq_h;        // sequence
12   sequencer           m_sequencer_h;  // sequencer
13   ahb_lite_agent_cfg  m_cfg_h;        // VIP configuration
14   ral                 m_ral_h;        // RAL
15
16   uvm_status_e status;
17
18   // Method prototypes
19   extern function
20       new(string name = "test", uvm_component parent);  // Class constructor
21   extern virtual function void
22       build_phase(uvm_phase phase);                     // Create component objects
23   extern virtual task
24       run_phase(uvm_phase phase);                       // Run phase
25   extern virtual function void
26       start_of_simulation_phase(uvm_phase phase);       // Get items from uvm_config_db
27 endclass : test
28
```

KUVA 39. Testiluokka

Testiluokan *build_phase*-metodiin lisättiin VIPin konfigurointi olion luominen, haettiin sille protokollaväylän ilmentymä *uvm_config_db*-tietokannasta ja asetettiin konfigurointi olio *uvm_config_db*-tietokantaan VIPin agenttia varten. VIPin konfiguraatioluokassa oli jo määriteltynä oletusarvot. Metodissa *start of simulation phase* haettiin RALin olion kahva *uvm_config_db*-tietokannasta ja lopuksi *run_phase*-metodissa tehtiin rekisterikirjoituksia testatakseen rekisteriympäristön toiminta sekä ajettiin simulaatio. Kuvassa 40 esitetään testiluokan metodien määritykset.

```
// Function: new
//
// - Creates test objects
function test::new(string name = "test", uvm_component parent);
  super.new(name,parent);
endfunction : new

// virtual function void: build_phase
function void test::build_phase(uvm_phase phase);
 super.build_phase(phase);

  m_cfg_h = ahb_lite_agent_cfg::type_id::create("m_cfg_h");

  // Connect AHB Lite VIP agent's configuration object's ahb vif interface
  if (!uvm_config_db#(virtual ahb_lite_interface)::get(
      this, "", "i_ahb_lite_vip.ahb_if", m_cfg_h.ahb_lite_vif)) begin
      `uvm_fatal(get_full_name(), "Cannot get i_ahb_lite_vip.ahb_if from config_db")
  end
  // Set the AHB Lite master agent configuration object to config_db
  uvm_config_db#(ahb_lite_agent_cfg)::set(null, "", "cfg", m_cfg_h);

  // create objects
  m_env_h = env::type_id::create("m_env_h", this);
  m_seq_h = seq::type_id::create("m_seq_h", this);
endfunction : build_phase

// Virtual task: run_phase
//
// TODO this is used to construct a working TB
// - Prints Hello World!
task test::run_phase(uvm_phase phase);
  phase.raise_objection(this);

  #10
  m_seq_h.m_ctrl_signal_2 = 1;
  m_seq_h.m_ctrl_signal_3 = 0;
  m_seq_h.start(m_sequencer_h);
  #100
  m_seq_h.m_ctrl_signal_2 = 0;
  m_seq_h.m_ctrl_signal_3 = 1;
  m_seq_h.start(m_sequencer_h);
  #100

  // Test writes to registers
  m_ral_h.register_1.write(status, 1);
  m_ral_h.register_2.write(status, 1);
  m_ral_h.register_3.write(status, 0);
  m_ral_h.register_4.write(status, 5);
  m_ral_h.register_5.write(status, 1);

  `uvm_info ("test::run_phase","Hello World!",UVM_LOW);

  phase.drop_objection(this);
endtask : run_phase

// Function void: start_of_simulation_phase
function void test::start_of_simulation_phase(uvm_phase phase);

  // Get the sequencer object from config_db
  if (!uvm_config_db#(sequencer)::get(this, "", "m_sequencer_h", m_sequencer_h)) begin
    `uvm_fatal(get_full_name (), "Cannot get m_sequencer_h from config_db")
  end
  // Get the RAL object from config_db
  if (!uvm_config_db#(ral)::get(this, "", "ral", m_ral_h)) begin
    `uvm_fatal(get_full_name (), "Cannot get ral from config_db")
  end
endfunction : start_of_simulation_phase

`endif  // TEST_SVH_
```

KUVA 40. Testiluokan metodit

5.9 Testipenkin viimeistely

Lopuksi luotiin alustava tulostauluoluokka ympäristön viimeistelemiseksi. Tarkoituksena olisi ollut myös luoda tulostauluoluokalle logiikka tarkistamaan DUT:n käyttäytymistä verrattuna sen odotettuun käyttäytymiseen sekä testiluokat jokaista DUT:n ominaisuutta kohden.

Näiden lisäksi olisi pitänyt luoda konfiguraatioluokat ympäristölle ja agentille sekä kattavuuspisteet ja -ryhmät testeille. Kattavuuspisteitä ja -ryhmiä sekä konfiguraatioluokkia ei esitelty tämän opinnäytetyön aikana.

5.9.1 Tulostaululuokka

Ympäristön viimeistelemiseksi puuttui tässä vaiheessa enää tulostauluKuokka. Tulostaululuokka (kuva 41) rekisteröitiin tehtaalle *uvm_component_utils*-makrolla ja luotiin *uvm_analysis_imp_decl*-makrolla loppuliitteet luotaville *uvm_analysis_imp*-porttityyppisille porteille sekä niiden *write*-metodeille.

Jäsenmuuttujiksi luotiin *uvm_analysis_imp*-tyyppiset muuttujat kontrollisignaalien väylän agentin monitorin ja AHB-Lite protokolla-agentin monitorin TLM-porttien liittämistä varten ja metodissa *build_phase* luotiin niille oliot *new*-metodilla. Jäsenmuuttujiksi luotiin myös *logic*-tyyppisiä muuttujia, joita tarvittiin sekvenssi alkioiden datan säilyttämiseksi tulostauluokassa.

```
1  `ifndef SCOREBOARD_SVH_
2  `define SCOREBOARD_SVH_
3
4  class scoreboard extends uvm_scoreboard;
5    `timescale 1ns/1ps
6    `uvm_component_utils(scoreboard)    // Register the scoreboard to the factory
7    `uvm_analysis_imp_decl(_control)    // TLM implementation port declaration for control items
8    `uvm_analysis_imp_decl(_ahb_lite)   // TLM implementation port declaration for ahb_lite items
9
10   // Class members
11   // TLM ports
12   uvm_analysis_imp_control #(seq_item, scoreboard)           m_analysis_imp_control;   // control signal TLM imp port
13   uvm_analysis_imp_ahb_lite #(ahb_lite_seq_item, scoreboard) m_analysis_imp_ahb_lite;  // ahb lite TLM imp port
14
15   // variables for control signal values
16   logic m_ctrl_signal_1;
17   logic m_ctrl_signal_2;
18   logic m_ctrl_signal_3;
19   logic m_ctrl_signal_4;
20   logic m_ctrl_signal_5;
21   logic m_ctrl_signal_6;
22
23   logic register_1;
24   logic register_2;
25   //
26   // ... rest of the variables/signals for the scoreboard and DUT checking here ...
27   //
28
29   // Method prototypes
30   extern function
31       new(string name = "scoreboard", uvm_component parent);     // Class constructor
32   extern virtual function void
33       build_phase(uvm_phase phase);                              // Used to build TLM ports and virtual interfaces
34   extern virtual task
35       run_phase(uvm_phase phase);                                // Implements checking of the DUT's behaviour
36   extern virtual function void
37       write_receiver(seq_item seq_item_h);                       // Defines,what is done to control sequence items
38   extern virtual function void
39       write_ahb_lite(ahb_lite_seq_item ahb_lite_seq_item_h);     // Defines,what is done to ahb lite sequence items
40 endclass : ast_sb
41
42 // Function: new
43 function scoreboard::new(string name = "scoreboard", uvm_component parent);
44   super.new(name, parent);
45 endfunction : new
46
47 // Virtual function void: build_phase
48 function void scoreboard::build_phase(uvm_phase phase);
49   super.build_phase(phase);
50
51   // Build TLM ports
52   m_analysis_imp_control  = new("m_analysis_imp_control", this);
53   m_analysis_imp_ahb_lite = new("m_analysis_imp_ahb_lite", this);
54 endfunction : build_phase
```

KUVA 41. Tulostaululuokka ja *new*- sekä *build_phase*-metodi

Tulostaululuokalle luotiin *write*-metodit vastaanottamaan sekvenssi alkioita kontrollisignaalien väylän agentin ja AHB-Lite VIP protokolla-agentin monitoreilta. Metodeille annettiin *uvm_analysis_imp_decl*-makroilla määritetyt loppuliitteet niiden nimien perään.

Metodissa *write_control* (kuva 42) tulostettiin funktion alussa sekvenssi alkion sisältämä data *print*-metodilla, joka mahdollistettiin, kun rekisteröitiin sekvenssi alkio -luokassa *uvm_field*-makroilla jokainen jäsenmuuttuja. Tulostuksen lisäämisen vuoksi, monitoriluokasta voitiin poistaa väliaikaiset tulostukset väylän arvoista. Lopuksi metodissa siirrettiin sekvenssi alkion data tulostaululuokan jäsenmuuttujiin.

Metodissa *write_ahb_lite* (kuva 42) tulostettiin sekvenssi alkion data ja määritettiin ehtolauseet, jotka tarkastelivat, oliko kyseessä rekisterikirjoitus vai -lukutapahtuma. Näiden ehtolauseiden sisään luotiin puolestaan *case*-ehtolauseet, jotka analysoivat rekisterin osoitteen ja siirsivät sen perusteella sekvenssi alkion datan oikeaan muuttujaan sekä tulostivat tiedot, mikä rekisteri oli kyseessä ja, mitä rekisteriin kirjoitettiin, jos kyseessä oli kirjoitustapahtuma. Tulostauluokan TLM-portit voitiin seuraavaksi liittää ympäristössä agenttien monitoreihin.

```
55
56  // Virtual task: run_phase
57  task scoreboard::run_phase(uvm_phase phase);
58
59  //
60  // ... Scoreboard logic goes here ...
61  //
62
63  endtask : run_phase
64
65  // Virtual function void: write_receiver
66  //
67  // - This function is called everytime the write-method is called in monitor.svh-class
68  // - Stores data from the seq_item_h into variables
69  function void scoreboard::write_control(seq_item seq_item_h);
70    `uvm_info(get_full_name(), "New sequence item received", UVM_HIGH);
71    `uvm_info(get_full_name(),seq_item_h.sprint(),UVM_DEBUG); // prints at every positive clk edge
72
73    m_ctrl_signal_1 = seq_item_h.m_ctrl_signal_1;;
74    m_ctrl_signal_2 = seq_item_h.m_ctrl_signal_2;
75    m_ctrl_signal_3 = seq_item_h.m_ctrl_signal_3;
76    m_ctrl_signal_4 = seq_item_h.m_ctrl_signal_4;
77    m_ctrl_signal_5 = seq_item_h.m_ctrl_signal_5;
78    m_ctrl_signal_6 = seq_item_h.m_ctrl_signal_6;
79  endfunction : write_receiver
80
81  // Virtual function void: write_ahb_lite
82  //
83  // - This function is called everytime the write-method is called in VIP monitor
84  // - Stores data from the ahb_lite_seq_item into variables
85  // - Generates a warning if a write or read is done to an address which does not correspond to the DUT's address space
86  function void scoreboard::write_ahb_lite(ahb_lite_seq_item ahb_lite_seq_item_h);
87
88    // Print data by using uvm_sequence_item class method
89    ahb_lite_seq_item_h.print();  // verbosity: UVM_HIGH
90
91    // Check if sequence was a write operation
92    if (ahb_lite_seq_item_h.op_type == WRITE) begin
93      case (ahb_lite_seq_item_h.address)
94        'h00    : begin
95                    register_1 = ahb_lite_seq_item_h.data;
96                    `uvm_info(get_full_name(), $sformatf("Register write to register_1: %0d", register_1),UVM_LOW)
97                  end
98        'h04    : begin
99                    register_2 = ahb_lite_seq_item_h.data;
100                   `uvm_info(get_full_name(), $sformatf("Register write to register_2: %0d", register_2), UVM_LOW)
101                 end
102       // ... rest of the registers ...
103       default : `uvm_warning(get_full_name(), $sformatf("Received address %0d outside of address space",
104                       ahb_lite_seq_item_h.address))
105     endcase
106   end
107   if (ahb_lite_seq_item_h.op_type == READ) begin
108     case (ahb_lite_seq_item_h.address)
109     // ... Reads ...
110     endcase
111   end
112 endfunction : write_ahb_lite
113
114 `endif  // SCOREBOARD_SVH_
115
```

KUVA 42. Tulostaulu-luokan *write*-metodit

5.9.2 Ympäristöluokka

Ympäristöluokkaan lisättiin tulostaululuokka tyyppinen muuttuja sekä luotiin sille olio *build_phase*-metodissa UVM-tehtaan metodilla. Ympäristöluokalle (kuva 43) luotiin vielä *connect_phase*-metodi, jossa liitettiin AHB-Lite protokolla-agentin sekä kontrollisignaalien väylää varten luodun agentin monitorin TLM-portit tulos-taululuokan *imp*-porttityyppisiin portteihin *connect*-metodilla.

```
`ifndef ENV_SVH_
`define ENV_SVH_

class env extends uvm_env;
  `uvm_component_utils(env)

  // Class members
  agent      m_agent_h;       // agent
  reg_env    m_reg_env_h;     // register environment
  scoreboard m_scoreboard_h;  // scoreboard

  // Method prototypes
  extern function
      new(string name = "env", uvm_component parent);  // Class constructor
  extern virtual function void
      build_phase(uvm_phase phase);                     // Used to build components
  extern virtual function void
      connect_phase(uvm_phase phase);                   // Used to connect components' TLM ports
endclass : env

// Function: new
function env::new(string name = "env", uvm_component parent);
  super.new(name, parent);
endfunction : new

// Virtual Function: build_phase
function void env::build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Create objects via factory and store objects into handles
  m_agent_h      = agent::type_id::create("m_agent_h", this);
  m_reg_env_h    = reg_env::type_id::create("m_reg_env_h", this);
  m_scoreboard_h = scoreboard::type_id::create("m_scoreboard_h", this);
endfunction : build_phase

// Virtual Function: connect_phase
function void env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  // Connect the receiver Agent's TLM port to scoreboard's TLM imp
  m_agent_h.m_agent_analysis_port.connect(m_scoreboard_h.m_analysis_imp_control);
  // Connect the ahb lite Agent's TLM port to scoreboard's TLM imp
  m_reg_env_h.m_ahb_lite_analysis_port.connect(m_scoreboard_h.m_analysis_imp_ahb_lite);
endfunction : connect_phase

`endif  // ENV_SVH_
```

KUVA 43. Valmis ympäristöluokka

6 JOHTOPÄÄTÖKSET JA POHDINTA

Testipenkkiä ei saatu tehtyä loppuun saakka, mutta suurin osa esittämättömistä testipenkin toteutuksen osista ei olisi voitu esittää työssä, koska ne sisältäisivät luottamuksellista tietoa. Testipenkin toteuttamisesta jäivät DUT:n käyttäytymisen verifioiminen tulostauluoluokassa, testiluokat jokaista DUT:n toimintoa tai ominaisuutta kohden ja konfigurointiluokkien luominen ympäristö- sekä agenttiluokille. Näiden lisäksi olisi pitänyt luoda testiluokkiin *coverpoint*- ja *covergroup*-tyyppisiä muuttuja, joita käytettäisiin keräämään kattavuuteen liittyviä tietoja. UVM-testipenkin toteuttamisen aikana opittiin kuitenkin paljon ja opitun pohjalta osataan viedä testipenkin toteutus loppuun saakka.

Haasteena UVM:n opettelussa oli sen käyttäminen oikeaoppisesti. Käyttäjän tulisi tiedostaa ja käyttää kantaluokkakirjaston metodeja, muuttujia ja luokkia seuratakseen metodologiaa, mutta niiden oppimiseksi täytyy syventyä kantaluokkien sisältöön. UVM:n käyttöä opeteltiin lukemalla UVM:n standardia, itse kantaluokkakirjaston lähdekoodeja sekä useita verkkosivuja. Ohjelmointikielien kirjastoja käytetään helpottamaan ja nopeuttamaan työtä tarjoamalla valmiita metodeja käytettäväksi, jotka suorittavat toistuvia tai monimutkaisia toimenpiteitä. Toisin kuin ohjelmointikielien kirjastot, UVM:n kantaluokkakirjastoa on tarkoituksena käyttää aina kun mahdollista metodologian vuoksi.

UVM:n kantaluokkien metodien ja muuttujien lisäksi täytyy myös opetella UVM:n oikeanlainen käyttö. Esimerkiksi UVM-komponentit voisi luoda ympäristön ja agentin sijan suoraan testiluokkaan. Tästä syystä testipenkin toteuttamisessa on helppo poiketa metodologiasta käyttämällä ainoastaan SystemVerilogin ominaisuuksia. UVM on myös joustava ja tarjoaa useita tapoja toteuttaa testipenkkejä käyttämällä metodologiaa. Esimerkiksi tulostauluoluokkaan voisi *imp*-tyyppisten porttien sijan luoda *export*-tyyppisiä portteja, jotka liitettäisiin FIFO-puskureihin. Useiden tapojen väliltä valitseminen vaatii tekijältä jo asiantuntijuutta ja pitkäaikaisempaa kokemusta.

Testipenkin toteutuksen aikana arvioitiin myös mahdollisuutta automatisoida sen luominen. Testipenkkien toteuttaminen voitaisiin opitun perusteella automatisoida ainakin tiettyyn pisteeseen saakka, koska UVM tarjoaa standardin tavan toteuttaa niitä ja samanlaiset toimenpiteet toistuvat luokissa verifioitavasta lohkosta riippumatta. Koko UVM-testipenkin toteutusta ei kuitenkaan ole välttämättä mahdollista automatisoida verifioitavien lohkojen ainutlaatuisuuden takia. Varsinkin tulostaulu- ja testiluokkia olisi haastava automatisoida, koska ne toteutetaan yleensä DUT:lle räätälöidysti, eivätkä sisällä samankaltaisia toimenpiteitä lohkosta toiseen.

LÄHTEET

Aho, A., Lam, M., Sethi, R. & Ullman, J. 2007. Compilers. Principles, Techniques, & Tools. 2. painos. Boston: Pearson Education, Inc.

AMBA 3 AHB-Lite Protocol Specification. 2006. Spesifikaatio. Päivitetty 6.6.2006. Luettu 15.4.2021.
https://developer.arm.com/documentation/ihi0033/a/Introduction/About-the-protocol/Slave

Connecting register env. n.d. Verkkosivu. Luettu 14.3.2021.
https://www.chipverify.com/uvm/connecting-register-env

Driver Sequencer Handshake. n.d. Verkkosivu. Luettu 7.3.2021.
https://www.chipverify.com/uvm/uvm-driver-sequencer-handshake

EDA playground. n.d. Simulaattori. Luettu 16.4.2021.
https://www.edaplayground.com/

Golson, S. & Clark, L. 2016. Language wars in the 21st century: Verilog versus VHDL-revisited. Artikkeli. Luettu 21.1.2021.
https://trilobyte.com/pdf/golson_clark_snug16.pdf

Header guards. 2016. Verkkosivu. Julkaistu 5.5.2016. Päivitetty 26.12.2020. Luettu 10.2.2021.
https://www.learncpp.com/cpp-tutorial/header-guards/

IEEE 1800.2-2020. 2020. IEEE Standard for Universal Verification Methodology Language Reference Manual. Standardi. Julkaistu 14.9.2020. Vaatii tunnistautumisen. Luettu 16.1.2021.
https://ieeexplore.ieee.org/servlet/opac?punumber=9195918

IEEE 1800-2017. 2018. IEEE Standard for SystemVerilog. Unified Hardware Design, Specification, and Verification Language. Standardi. Julkaistu 22.2.2018. Vaatii tunnistautumisen. Luettu 16.1.2021.
https://ieeexplore.ieee.org/servlet/opac?punumber=8299593

Modelsim. n.d. Verkkosivu. Luettu 16.4.2021.
https://eda.sw.siemens.com/en-US/ic/modelsim/

Moore, G. 1965. Cramming more components onto integrated circuits. Artikkeli. Luettu 14.4.2021.
https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf

Register Layer. n.d. Verkkosivu. Luettu 14.3.2021.
https://www.chipverify.com/uvm/register-layer

Rintala, M. & Jokinen, J. 2005. Olioiden ohjelmointi C++:lla. 4. painos. Helsinki: Talentum oyj.

Spear, C. 2008. SystemVerilog for Verification. 2. painos. New York: Springer Science+Business Media, LLC.

SystemVerilog Interface. n.d. Verkkosivu. Luettu 17.2.2021. https://www.chipverify.com/systemverilog/systemverilog-interface

SystemVerilog package. n.d. Verkkosivu. Luettu 11.2.2021. https://www.chipverify.com/systemverilog/systemverilog-package

TLM Implementation Port Declaration Macros. n.d. Verkkosivu. Luettu 14.3.2021. https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/macros/uvm_tlm_defines-svh.html

Using get_next_item(). n.d. Verkkosivu. Luettu 13.3.2021. https://www.chipverify.com/uvm/uvm-using-get-next-item

Using get() and put(). n.d. Verkkosivu. Luettu 13.3.2021. https://www.chipverify.com/uvm/driver-using-get-and-put

UVM Agent | uvm_agent. n.d. Verkkosivu. Luettu 13.3.2021. https://www.chipverify.com/uvm/uvm-agent

UVM Common Phases. n.d. Verkkosivu. Luettu 4.3.2021. https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/base/uvm_common_phases-svh.html#uvm_build_phase

uvm_config_db Examples. n.d. Verkkosivu. Luettu 1.3.2021. https://www.chipverify.com/uvm/uvm-config-db-examples

UVM Config db. n.d. Verkkosivu. Luettu 1.3.2021. https://verificationguide.com/uvm/uvm-configuration-database/

UVM Environment [uvm_env]. n.d. Verkkosivu. Luettu 16.3.2021. https://www.chipverify.com/uvm/uvm-environment

UVM Factory Override. n.d. Verkkosivu. Luettu 3.3.2021. https://www.chipverify.com/uvm/uvm-factory-override

UVM Introduction. n.d. Verkkosivu. Luettu 26.3.2021. https://www.chipverify.com/uvm/uvm-introduction

UVM Monitor [uvm_monitor]. n.d. Verkkosivu. Luettu 13.3.2021. https://www.chipverify.com/uvm/uvm-monitor

UVM Phases. n.d. Verkkosivu. Luettu 4.3.2021. https://www.chipverify.com/uvm/uvm-phases

UVM RAL Adapter. n.d. Verkkosivu. Luettu 17.3.2021. https://verificationguide.com/uvm-ral/uvm-ral-adapter/

UVM RAL Predictor. n.d. Verkkosivu. Luettu 17.3.2021. https://verificationguide.com/uvm-ral/uvm-ral-predictor/

UVM Register Environment. n.d. Verkkosivu. Luettu 14.3.2021. https://www.chipverify.com/uvm/uvm-register-environment

UVM Register Model. n.d. Verkkosivu. Luettu 14.3.2021. https://www.chipverify.com/uvm/uvm-register-model

UVM Run-Time Phases. n.d. Verkkosivu. Luettu 4.3.2021. https://verificationacademy.com/verification-methodology-refe-rence/uvm/docs_1.1d/html/files/base/uvm_runtime_phases-svh.html

UVM Scoreboard. n.d. Verkkosivu. Luettu 14.3.2021. https://www.chipverify.com/uvm/uvm-scoreboard

UVM Sequence [uvm_sequence]. n.d. Verkkosivu. Luettu 6.3.2021. https://www.chipverify.com/uvm/uvm-sequence

UVM_Sequence_item. n.d. Verkkosivu. Luettu 6.3.2021. https://verificationguide.com/uvm/uvm-sequence-item/

UVM Test. n.d. Verkkosivu. Luettu 6.3.2021. https://www.chipverify.com/uvm/uvm-test

VCS. n.d. Verkkosivu. Luettu 16.4.2021. https://www.synopsys.com/verification/simulation/vcs.html

Verilog Task. n.d. Verkkosivu. Luettu 12.2.2021. https://www.chipverify.com/verilog/verilog-task

Verilog Timescale. n.d. Verkkosivu. Luettu 14.2.2021. https://www.chipverify.com/verilog/verilog-timescale

Questa Advanced Simulator. n.d. Verkkosivu. Luettu 16.4.2021. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

Xcelium Logic Simulation. n.d. Verkkosivu. Luettu 16.4.2021. https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/si-mulation-and-testbench-verification/xcelium-simulator.html