
TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

# DIPLOMITYÖ

## Muunneltavan FPGA-testiympäristön käyttö reaaliaikaisessa RTL-verifioinnissa


| | |
|---|---|
| Tekijä | Janne Paakkolanvaara |
| Valvoja | Jukka Lahti |
| Toinen tarkastaja | Timo Rahkonen |
| (Työn tekninen ohjaaja | Teemu Sirviö) |


Toukokuu 2018

**Paakkolanvaara J. (2018) Muunneltavan FPGA-testiympäristön käyttö reaaliaikaisessa RTL-verifioinnissa.** Oulun yliopisto, Sähkötekniikan tutkinto-ohjelma. Diplomityö, 41s. Ohjaaja: Jukka Lahti.

# TIIVISTELMÄ

**Prototypointi on kasvattanut osaansa ASIC-piirien suunnittelussa viimeisen kahden vuosikymmenen aikana RTL-suunnitelmien kasvaessa valtavasti. Prototypoinnissa voidaan varmentaa suunnitelmaa reaalimaailmassa ennen, kuin se valmistetaan piille. Tämä säästää valtavasti kustannuksia, sillä ASIC-piirin uudelleen valmistaminen on erittäin kallista. Prototypoinnissa voidaan myös aloittaa ohjelmistokehitys ennen kuin varsinainen piille painettu piiri valmistuu. Tämä nopeuttaa piirin markkinoille pääsyä huomattavasti. Prototypointia tehdään tyypillisesti emulaattorilla ja tässä työssä keskiössä olevilla FPGA-levyillä.**

**Tässä työssä esitellään muunneltavan FPGA-testiympäristön haasteita RTL-suunnitelman reaaliaikaisessa verifioinnissa. Työssä rakennetaan testipenkin muistin kirjoittajasta sekä lukijasta muunneltava yleismoduulipari. Näihin moduuleihin luodaan valmiudet tukemaan reaaliaikaista FPGA-testaamista. Työssä perehdytään tuntemattomien kellotaajuuksien välisen datansiirron hallintaan ja parametreilla muunneltavan testikomponentin ylläpitoon. Työssä on tavoitteena saada datansiirto toimimaan virheettömästi riippumatta kellojen taajuuksista. Työssä rakennetaan testipenkkisovellus käyttäen moduulien reaaliaikaominaisuuksia. Lopuksi katsastellaan tämän testipenkkisovelluksen kypsymistä RTL-verifioinnissa ja minkälainen osa tässä on prototyypin ylläpitäjällä.**

**Avainsanat: FPGA, RTL, Kelloalue.**

**Paakkolanvaara J. (2018) Use of a reconfigurable FPGA test environment in real-time RTL verification.** University of Oulu, Degree Programme in Electrical Engineering. Master's Thesis, 41p. Director: Jukka Lahti.

## ABSTRACT

**Prototyping has grown its part in the design of ASIC-chips in the last two decades as the RTL-designs have grown enormously. Prototyping enables the verification of said design in the physical world before it is put into silicon. This saves money, as making an ASIC-chip is very expensive. Prototyping can also be harnessed for the purposes of software development before the target chip is complete. Doing this improves the chip's time-to-market, which is crucial in making a profit from said chip. Prototyping is typically done on emulators and FPGA-boards, latter being the main focus of this work.**

**This work showcases the challenges involved in real-time testing of an RTL-design on a reconfigurable FPGA testbench. The focus of the work is to solve a usual issue with said environment: sufficient data transfer speed over clock domains. The topics include managing data transfers over multiple previously unknown clock domains and maintaining a reconfigurable test component. The goal of the work is to create a bridge between two unknown clock domains. An example of the use of the real-time configuration is shown. Finally, a peek is taken into the maturation of the RTL-verification solution and what is required for the one in charge of maintaining the prototype.**

**Keywords FPGA, RTL, Clock domain.**

# SISÄLLYSLUETTELO

# LYHENTEIDEN JA MERKKIEN SELITYKSET

| | |
|---|---|
| ASIC | Sovelluskohtainen mikropiiri. |
| A/D-muunnin | Laite, joka muuntaa analogisen signaalin digitaalisiksi arvoiksi. |
| CDC | Kelloalueensiirto |
| DPD | Digitaalinen esisärötin |
| DSP | Digitaalinen signaaliprosessori |
| D/A-muunnin | Laite, joka muuntaa digitaaliset arvot analogiseksi signaaliksi. |
| FIFO | Ensimmäinen sisään ensimmäinen ulos muistirakenne |
| FPGA | Kenttäohjelmoitava mikropiiri, jonka logiikka voidaan ohjelmoida uudelleen. |
| FSM | Tilakone |
| IP | Immateriaalioikeus |
| JESD | Xilinx digitaalinen ADC rajapinta. |
| Kelloalue | Alue, mitä yksi oskillaattori hallitsee digitaalisella piirillä. |
| MMCM | Sekatilainen kellonhallitsija |
| NRT | Ei reaaliaikainen |
| PLL | Vaihelukittu silmukka |
| RAM | Hajapääsymuisti |
| RT | Reaaliaikainen |
| RTL | Rekisteritason kuvaus. |
| Synteesi | Rekisteritason kuvauksen muuntaminen logiikkaporteiksi. |

# 1. JOHDANTO

Sovelluskohtaisia integroituja piirejä, eli ASIC-piirejä käytetään sovelluksiin, joissa tarvitaan tietyn tapaista laskentatehoa. Sovelluskohtaisia kohteita on esimerkiksi tukiasemien piirit, joiden koodi, eli RTL-suunnitelma täytyy varmentaa erittäin tarkasti, sillä ASIC-piirin valmistaminen on hyvin kallista. Varmennus täytyy olla ripeää, sillä alalla vaaditaan menestymiseen nopeaa piirin saattamista spesifikaatiosta tuotteeseen.

FPGA-ympäristöjä käytetään RTL-suunnittelussa prototypointiin, jossa luodaan FPGA-levylle ympäristö, joka toimii kuten valmis ASIC. Tämä ympäristö toimii usein huomattavasti hitaammin kuin valmis piiri, mutta se luo pohjan ohjelmistokehityksen aloittamiseen hyvissä ajoin ennen kuin ASIC on valmis. Tarve on kuitenkin myös reaaliaikaisille sovelluksille, joilla voidaan varmentaa piirin analogista rajapintaa. Näissä sovelluksissa tavoite on saada FPGA-ympäristö toimimaan ulkoapäin katsottuna ns. oikealla nopeudella eli reaaliaikaisesti. Reaaliaikaisuuden saavuttamisen haasteet, riittävä muistin nopeus, suunnitelman ajoituksiin saaminen, sekä riittävä ja luotettava kelloalueiden välinen datan siirtonopeus ovat aina läsnä. Työn FPGA-testipenkissä käytetään eri prototyyppien erilaisiin vaatimuksiin muunneltavia moduuleita, jotka toimivat niille annettujen parametrien mukaan. Muunneltavien moduulien ohjelmoinnissa reaaliaikaisuus huomioidaan pitämällä suunnitelman rekistereiden välillä mahdollisimman vähän kombinaatiologiikkaa, joka auttaa synteesin ajoitukseen saamista.

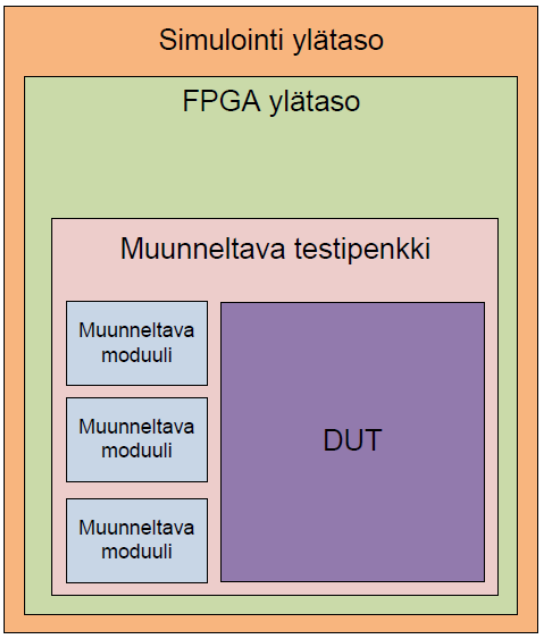
Keskeinen ongelmien aihe on digitaalitekniikassa monesti vastaantuleva kelloalueen vaihtaminen ja sen aiheuttamat haasteet ko. sovelluksen suunnittelussa. Työssä esitellään, miten testipenkin muunneltavaan yleismoduuliin lisätään konfiguraatioita ja siihen liittyviä haasteita FPGA-ympäristössä. Synteesityökaluja ja niiden käyttöä ei työssä käsitellä, mutta niiden vaikutus päätöksiin mainitaan. Kohteena on rakentaa mainittuun valmiiseen, ei vielä muunneltavaan yleismoduuliin toiminnallisuutta, jolla reaaliaikainen prototypointi olisi mahdollista. Tässä työssä edellä mainittu reaaliaikainen sovellus rakennetaan osaksi muunneltavaa FPGA-testipenkkiä, joka lopulta käännetään kahden eri nopeusluokan ja mallin FPGA-levyn ympäristöön. Työn sovellus rakennetaan näistä kahdesta hitaammalle levylle, ja koska niiden erojen aiheuttamat ajoitushaasteet ovat puhtaasti jo valmiiksi alustetun nopeamman levyn puolella, ei niitä tarvitse tässä työssä ratkaista. Työn ratkaistavia ongelmia kellotaajuuden reaaliaikaiseksi saattamisen, sekä yleismoduulien toimivuuden muissa prototyypeissä varmistamisen lisäksi on työn kahden FPGA-levyn välisien yhteyksien minimoiminen, sillä niiden suuri määrä vähentää ympäristön suorituskykyä.

OULUN YLIOPISTO
UNIVERSITY of OULU

Kuva 1. Yliopiston logo

# 2. MUUNNELTAVAVUUS JA REAALIAIKAISUUS

Prototypoinnissa varmennetaan RTL-suunnitelmaa käyttämällä FPGA-ympäristöjä. Vaikka on olemassa ASIC-piirejä suurempia FPGA-ympäristöjä, tässä työssä käytetyt ympäristöt ovat yksinkertaistamisen, nopeuden, sekä kustannustehokkuuden takia kuitenkin huomattavasti pienempiä transistoriluvultaan kuin valmiit ASIC-piirit, jolloin voidaan testata vain osaa alijärjestelmistä kerrallaan. Turhan työn minimoimiseksi on siis hyödyllistä ylläpitää yhtä kuvan 2 mukaista muunneltavaa testipenkkiä, jota voidaan hyödyntää kaikissa eri prototyypeissä ja simulaatioissa. Muunneltavaan testipenkkiin kuuluu myös muunneltavat yleiskomponentit, joiden ominaisuuksia muutetaan RTL-tason parametrimuutoksilla.




Kuva 2. Muunneltavan testipenkin hierarkia.

## 2.1. Muunneltavuuden säilyttäminen

Jokainen prototyypin ylläpitäjä päivittää yhteiseen testipenkkiin kaiken, mitä itse tarvitsee. Testipenkki tukee reaaliaikaisia prototyyppejä, joissa kellotaajuus on sama kuin ja/tai käyttäytyy ulkoisesti samalla tavalla kuin valmiilla ASIC-piirillä, sekä ei-reaaliaikaisia prototyyppejä, joissa kellotaajuus on murto-osa oikeasta. Tämän täytyy onnistua testipenkin olematta kuitenkaan kovin suuri. Tätä tuetaan muunneltavilla testipenkkimoduuleilla, jotka kääntyvät eri tavalla eri parametreilla. Kyseenomaisten osien ohjelmoiminen on kuitenkin haastavaa, sillä uusi ominaisuus ei saa vaikuttaa muiden konfiguraatioiden toimintaan.

Verilog 2005-kielellä ohjelmoidussa testipenkissä muunneltavuutta tukee define-rakenne, jolloin testipenkkiin voi käännöksen aikana sisällyttää vain tarvittavat osat. Käytännössä tällä on kuitenkin heikkoutena nopeasti kasvava monimutkaisuus, joka tekee testipenkin käyttöönotosta hidasta. Ehkäisevänä toimena käytetään yhtenäisiä ohjelmointikäytäntöjä sekä kaiken tarpeettoman poistamista. [1]

Suuren yhtenäisen testipenkin etuja on kuitenkin yksittäisten prototyyppien vastuun vaihtamisen helppous ja päällekkäisen työn minimoiminen. Jokainen suunnittelija voi tehdä useamman alijärjestelmän prototyypin samalla ympäristöllä. Tämän lisäksi testipenkki vaatii kerrallaan vain yhden työntekijän panoksen päivittämiseen.

### *2.1.1. Syntetisoituvuuden säilyttäminen*

Koska testipenkkiä käytetään pääasiallisesti FPGA-levyllä, pitää sen syntetisoitua. Tämän varmistamiseksi ei aina riitä yksi työkalu, sillä organisaatiossa niitä voi olla monia. Jokaisen konfiguraation ei kuitenkaan tarvitse olla syntetisoituva, sillä ainoastaan simuloinnissa käytettäviä konfiguraatioita ei käännetä koskaan FPGA-levylle. Syntetisoituvuuden saavuttamista helpottaa oikeanlaisen kielen käyttö. VHDL on vahvasti tyypitetty kieli, jossa on hyvin vähän rakenteita, joita synteesityökalut eivät tue. Se on myös helposti luettavaa, jolloin sen ylläpitäminen on nopeaa siirtää suunnittelijalta toiselle. Kyseinen kieli on kuitenkin hyvin jäykkä, joka ei ole toivottava ominaisuus muunneltavan testipenkin ylemmille tasoille. Toimiva käyttökohde ovat pienet muunneltavat moduulit, joita pohjustetaan useampi ylemmän tason kielellä tehtyyn testipenkkiin. FPGA-testipenkin koodin vaatimukset eivät ole yhtä tiukat kuin ASIC-koodilla. Tämä johtuu siitä, että FPGA-levylle annetut rajoiteparametrit ovat löyhempiä. Koodin on kuitenkin oltava helposti luettavaa optimaalisen nopeuden/pinta-alan sijasta, mutta toiminnallisuuden on oltava vakio ja sen pitää pystyä syntetisoitumaan kohdelevylle. [2]

### *2.1.2. Olemassa olevien konfiguraatioiden ja toimintojen eheys*

Moduulissa, jonka vaatimukset muuttuvat testaajien tarpeiden mukaan tarvitsee normaalista RTL-suunnittelusta poikkeavan suunnitteluvuon. Moduuli luodaan alkuperäiseen tarpeeseen, johon lisätään ominaisuuksia prototyyppitestaajien tarpeiden mukaan. Uuden lisäämisessä on siis tärkeää varmistaa, ettei olemassa olevien konfiguraatioiden toiminnallisuus muutu. Tämä vaatii jokaisen konfiguraation varmentamisen joka kerta kun moduuliin tehdään korjaus tai lisätään uusi ominaisuus. Seurauksena on hyvin nopeasti laajeneva testien määrä, jolloin nopean ja pitkälle automatisoidun simulaatiovuon tärkeys kasvaa. Tämänlaisessa ohjelmoinnissa voidaan työtä nopeuttaa sekä helpottaa keskittämällä varmennus pääasiallisesti niihin tilanteisiin, missä testaajat käyttävät moduulia.

### *2.1.3. Testausominaisuudet ja niiden ylläpito*

Muunneltavaan moduuliin lisätään status-, sekä virhesignaaleja kertomaan testaajalle, miten moduuli käyttäytyy. Näitä muutaman bitin levyisiä signaaleja luodaan, koska FPGA-ympäristössä näkyvyyttä piirin sisälle ei ole tai se on hyvin rajattua. Nämä jaetaan kahteen kategoriaan: Moduulin käytöstä johtuvat virheet, kuten puskurin ylivuoto ja RTL-moduulien virhetilanteista johtuvat vikatilat, kuten näytteiden oikeellisuuden putoaminen kesken ajon. Signaalit valitaan ja rakennetaan siten, että ne ovat mahdollisimman ystävällisiä käyttäjälle sekä ohjelmoijalle. Helppoutta edistää muun muassa signaalien toiminnallisuuden vakinaistamisen sekä riippumattomuus käytetystä konfiguraatiosta. [5]

## 2.2. Erilaisten testialustojen käyttö

Uuden ominaisuuden ohjelmoinnissa ei aina ole suositeltavaa kääntää moduulia suoraan FPGA-piirille, sillä logiikan verifioinnissa tarvitaan täyttä näkyvyyttä moduulin sisälle. On siis hyödyllistä verifioida itse logiikka ja datapolut ennen synteesin käynnistämistä käyttäen työkaluja, joissa näkyvyys on hyvä. Muunneltavaa testipenkkiä voidaan käyttää useammalla eri alustalla, mutta jokainen alusta tarvitsee oman ylätasonsa, jonka sisälle testipenkki alustetaan.

### *2.2.1. Simulaattori*

Simulaattorin rooli FPGA-testipenkin verifioinnissa on varmistaa testipenkin logiikan ja perustoimintojen toimivuus käyttäen vähän resursseja ja aikaa. Synteesityökalut ovat kalliita, jolloin niiden lisenssejä on monesti vähän käytössä. On siis tavoiteltavaa saada loogisesti toimiva käännös heti ensiyrittämällä.

Käännetyn RTL-koodin ja testipenkin yhdistelmästä tarkistetaan kaikkien haluttujen osien mukana olo. Tämä tehdään yleisesti lukemalla moduulien tunnisterekisterit. On myös moduuleita, joilla ei tunnisterekisteriä ole, kuten testipenkin yleismoduulit. Näiden olemassaolo voidaan tarkistaa piirikaavionäkymästä, joka on ominaisuus lähes kaikissa simulaattoreissa. Simulaatiossa ei ole yleisesti järkevää varmentaa muuta kuin testipenkin ominaisuudet ja RTL-suunnitelman tunnisterekisterit.

Ongelmia simulaattorissa on ASIC-suunnitelman mukanaan tuoma valtava porttien määrä. Tästä johtuen monimutkikkaita ja pitkiä testejä on todella hidasta ajaa, sillä itse RTL-simuloiminen vaatii paljon resursseja tietokoneelta. Tästä syystä työssä on pääasiallisesti käytetty simulaattoria pelkästään datapolun testaamiseen erillisellä, muutaman moduulin kattavalla testipenkillä.

Simulaatioita käytetään kuitenkin FPGA-levyllä havaitun ongelman ratkaisemiseen, sillä vaikka RTL-simulointi on hidasta, saadaan sillä täysi näkyvyys. On nopeampaa tehdä yksi kahden päivän simulaatio, kuin kääntää FPGA-prototyyppi kymmeniä kertoja, sillä työssä havaittu käännösaika FPGA-ympäristöön on kahdesta kymmeneen tuntiin.

### *2.2.2. Emulaattori*

Emulaattori on kokoelma joko prosessoreita tai FPGA-piirejä, joilla on paljon näkyvyyttä piirin sisälle. Levyille käännetystä suunnitelmasta voidaan ottaa ulos paljon aaltoja. Tämä on selvä hyöty, sillä emulaattori ei vaadi montaa käännöstä, jotta saadaan tutkittua suurinta osaa signaaleista. Emulaattori myös toimii huomattavasti nopeammin kuin simulaattori, jolloin täysiä testejä voidaan ajaa kattavasti. Emulaattori on siis tehokas ei-reaaliaikaisten ominaisuuksien verifioimisessa.

Emulaattori ei kuitenkaan ole riittävä tämän työn vaatimuksiin, joten niitä ei käytetä tässä työssä. On kuitenkin hyvä tietää, että emulaattorilla voidaan varmentaa RTL-suunnitelmaa FPGA-prototyypin tapaisesti.

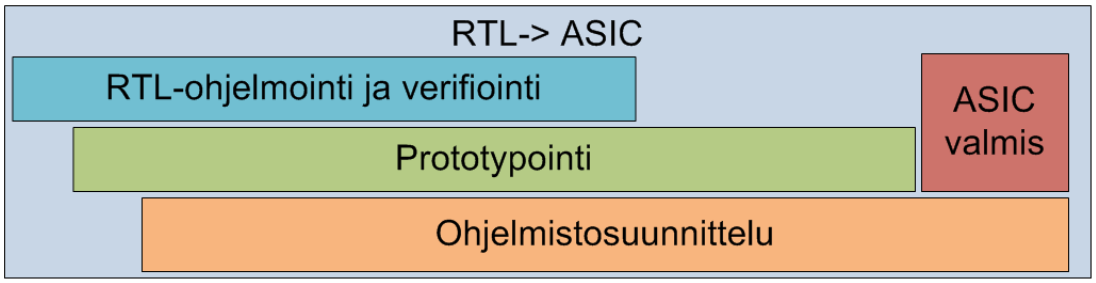
### *2.2.3. FPGA*

FPGA on digitaalinen muunneltava prosessori, johon voidaan ladata uusia konfiguraatioita nopeasti. FPGA-piiri on iso matriisi erilaisia loogisia elementtejä sekä muita digitaalisen prosessoinnin komponentteja, kuten DSP-lohkoja. Prototypoinnissa käytetään FPGA-levyjä, jotka ovat piirikortteja, joissa FPGA itsessään on. Nämä levyt toimivat kokonaisuutena, jossa on monia analogisia komponentteja, joiden kanssa itse piiri voi toimia. FPGA-levyjä voi myös olla useampi yhdessä FPGA-moduulissa, joka kasvattaa moduulin porttimäärää, mahdollistaen suurempien RTL-suunnitelmien varmentamisen.

FPGA-levyllä voidaan saavuttaa reaaliaikaisia sovelluksia digitaalisen ja analogisen välillä olevaan rajapintaan. Tämä on suuri hyöty, sillä näin voidaan rakentaa testiympäristö ASIC-piirin ympärille ennen sen valmistumista. Kuten ASIC-piireissäkin, on FPGA-piireissä valmistusprosessista johtuvia vaihteluita, näitä vaihteluita kutsutaan nopeusluokiksi. Nämä nopeusluokat voivat olla hidaste ja/tai haaste pyrkiessä reaaliaikaisiin sovelluksiin, sillä yhdellä nopeusluokka -2 FPGA-mallilla A saavutettu kellotaajuus voi olla mahdoton saman mallin A nopeusluokan -1 omaavassa piirissä. Tässä työssä käytetään kahta eri nopeusluokan ja mallin FPGA-levyä. [8]

Alustan suurin heikkous on näkyvyys, sillä käyttäjän on valittava ennen synteesiä signaalit, joita halutaan testin aikana tarkastella. Näitä signaaleja voidaan ottaa ulos levyltä vain rajattu määrä. Tämä yhdistettynä prototypoinnissa tyypillisiin pitkiin, yli 10 tunnin synteesiaikoihin tekee FPGA-ympäristöstä epäideaalisen logiikan varmentamisessa. Kyseinen heikkous aiheuttaa myös vaikeuksia prototyypin ylläpitäjälle, sillä saman prototyypin testaajilla on monesti yksilökohtaiset kiinnostavat signaalit.

FPGA-ympäristöjä käytetään laajojen ja monimutkaisten systeemitason testien ajamiseen nopeasti ja mahdollisuudet ovat päästä lähes reaaliaikaan. Yksinkertainen testi, jonka suorittamiseen menee simulaattorilla tunti, onnistuu FPGA-ympäristöllä alle sekunnissa. FPGA-ympäristöllä voidaan myös tehdä edellä mainitun testausympäristön lisäksi ASIC-piiriä tukevaa ohjelmistoa, joka onkin prototypoinnin yksi suuri etu. Ohjelmistosuunnittelun pitkälle vienti ennen ASIC-piirin valmistumista voi nopeuttaa piirin markkinoille lähettämistä yli vuodella. Kuvassa 3 prototypoinnin sijoitus ASIC-vuohon.

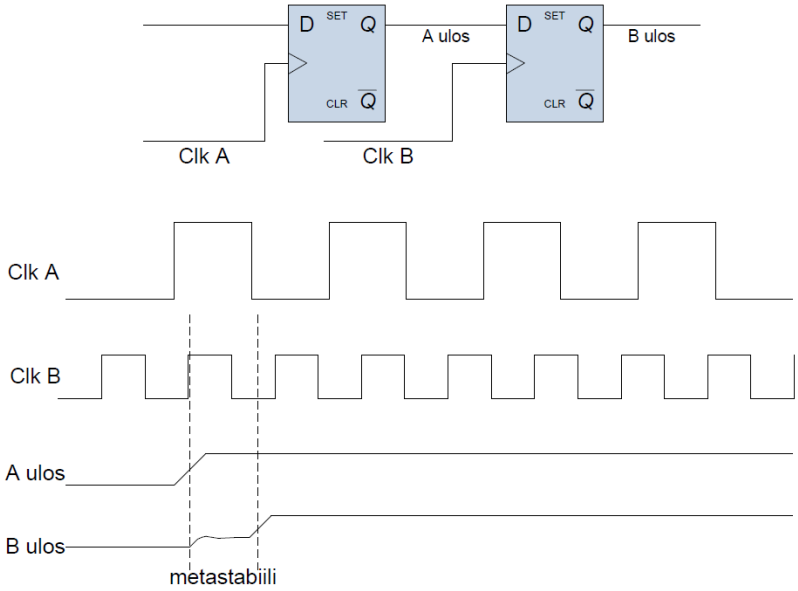



Kuva 3. Prototypoinnin rooli ASIC-vuossa.

## 2.3. Kelloaluesiirrot

Nykyaikaisissa mikropiireissä sekä FPGA-levyissä on monia kelloja, joka johtaa niiden välillä siirtymisen tarpeeseen. Syy, miksi piirissä löytyy kelloja, jotka toimivat eri kellotaajuudella on, että näin voidaan käyttää hidasta sekä nopeaa logiikkaa tehokkaasti samassa systeemissä. On logiikkaa, joka tarvitsee paljon kellosyklejä, mutta omaa vähän viivettä luovia tasoja, jolloin sitä kellotetaan lyhyellä syklillä. On myös logiikkaa, jossa on syvää kombinaatiologiikkaa, jolloin se omaa suuremman viiveen kahden pisteen välillä, näihin käytetään hitaita kelloja. Nämä logiikat kommunikoivat keskenään, jolloin tarvitaan kelloaluesiirtoja.

Suurin ongelma kelloaluesiirroissa on kuvassa 4 esitelty ilmiö, missä kiikun lähtö jää värähtelemään 0 ja 1 välille, eikä asetu kumpaankaan ennen seuraavaa nousevaa reunaa. Ilmiötä kutsutaan metastabiilisuudeksi. CDC-menetelmien tavoite on estää näiden tilojen aiheuttamat häiritsevät ominaisuudet, jotta voidaan luottaa piirin toimintaan ja tiedon oikeellisuuteen riippumatta, mitä osaa piiristä katselmoidaan. [3]




Kuva 4. Metastabiili ilmiö kelloalueen vaihdossa.

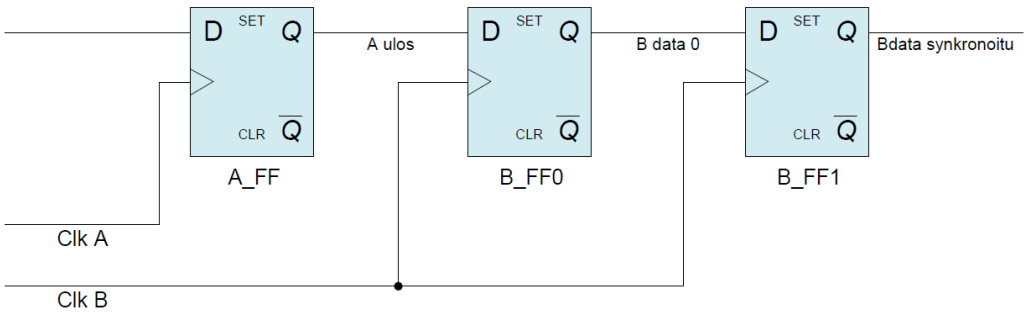
Metastabiilinen tila johtuu tulon ja sitä vastaanottavan kiikun kellojen epäsynkronisuudesta. Vastaanottavan kiikun kellon nousevan reunan osuessa keskelle vastaanottavan kiikun asettumisaikaa, voi vastaanottavan kiikun lähdöllä olla vaikeuksia päättää kumpaan tilaan asettua. Jos metastabiilisuus on mahdollista, varaudutaan siihen aina ASIC-ohjelmoinnissa. Sama pätee myös FPGA-ohjelmointiin. FPGA-piirillä kellotus on, matalatehoisia mikropiirejä lukuun ottamatta, hitaampaa kuin yleisissä ASIC-piireissä, jolloin metastabilisuuden hallinta on helpompaa. FPGA-levylle ohjelmoitaessa on kuitenkin oletettava metastabiilisuuden olevan mahdollista riippumatta kelloaluerajapinnan eri puolilla olevien kellojen taajuuksista. Mitä paremmin CDC hoidetaan FPGA-koodissa, sitä helpompaa on saavuttaa tiukempia kellorajoitteita, joka on tarpeellista, jos halutaan prototypoida ASIC-suunnitelmaa.

Prototypoinnin testipenkissä voidaan kuitenkin harvoissa tilanteissa toimia tämän ohjeistuksen vastaisesti. Nämä tilanteet ovat kuitenkin aina havaittuja käyttäytymisiä, eikä niitä tehdä rikkomaan CDC-sääntöjä alkuperäisessä suunnitelmassa. Esimerkkinä tästä on jokin hyvin hidas, noin 3 kertaa sekunnissa muuttuva signaali, joka ei aja minkäänlaista funktionaalista RTL-suunnitelman tai testipenkin logiikkaa, vaan on jokin virhe-, tai statussignaali, jolla etsitään virhetilanteiden syitä testipenkissä. Näissä signaaleissa ominaista on, että niitä tarkastellaan niin kauan, ettei metastabiilisuudesta ole haittaa. Kyseisessä tilanteessa on kuitenkin ymmärrettävä, että jos ei haluta nähdä signaalin tilan muuttumista mahdollisimman lähellä sitä kelloreunaa missä sen vaihto tapahtuu, ei ole syytä jättää luomatta signaalille CDC-rakennetta.

### *2.3.1. 2FF-kiikkusynkronisaattori*

Yleisesti kelloalueen siirrot tapahtuvat kahden kelloalueen välillä, jolloin niitä käsitellään lähde-, ja kohdealueina. Suunnittelija tietää molempien kelloalueiden taajuudet ennen suunnittelua, jolloin menetelmän valinnassa voidaan optimoida viiveet, pinta-ala-vaatimukset sekä rakenne.

Yleisesti signaalien metastabiilisuutta hallitaan kuvan 5 tapaisilla synkronisaattoreilla, missä vastaanottavalle kelloalueelle asetetaan jono kiikkuja. Vastaanottavan kelloalueen B_FF1 antaa B_FF0 lähdölle aikaa asettua joko nollaan tai yhteen. Pelkästään tämän yksinkertaisen metodin käyttö datan siirrossa kuitenkin vaatii sen, että Clk B kellotaajuus on vähintään 2 kertaa suurempi kuin Clk A. Tämä varmistaa sen, että B_FF0 näkee aina sisään tulevan signaalin. [3]



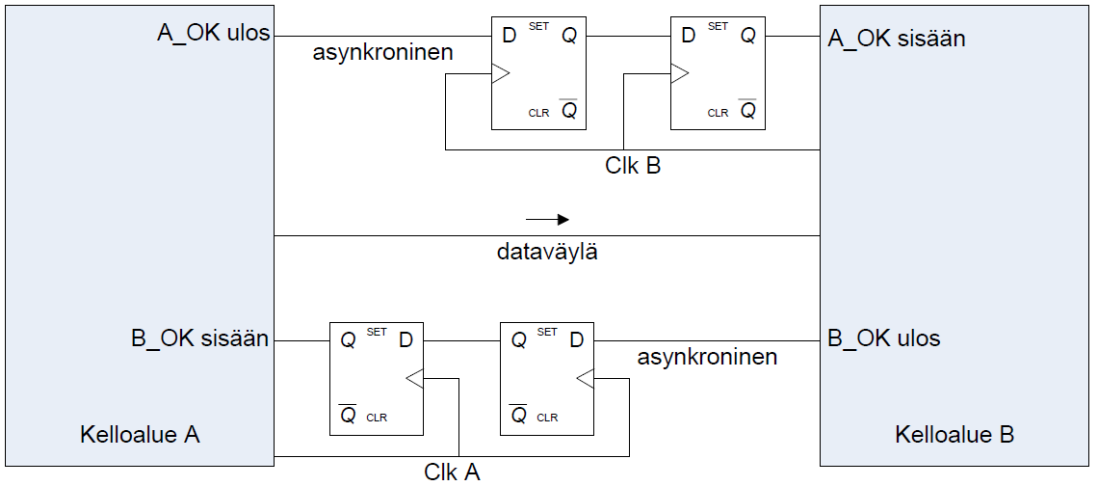

Kuva 5. Yksinkertainen 2FF-kiikkusynkronisaattori.

Tämäntapaiset synkronisaattorit ovat aina läsnä muissa metodeissa, sillä ne hallitsevat kelloaluesiirroista aiheutuvaa metastabilisuutta. Muut menetelmät pyrkivät estämään datan korruptiota ja keskittyvät datan siirtämiseen, kun kiikkusynkronisaattori poistaa siirron metastabiilisuuden aiheuttamat häiriöt muulle logiikalle. Tästä johtuen 2FF-synkronisaattoria pidetään CDC-menetelmien perusrakenteena. Synkronisaattoria ei välttämättä aina tarvita, mutta niiden poisjättäminen saattaa altistaa suunnitelman metastabiilisuudelle.

### *2.3.2. Kättelymenetelmä*

Kättelymenetelmä on varma, viivettä luova menetelmä datan siirtoon kelloalueiden välillä. Siinä lähettävä kelloalue A nostaa kättelysignaalin A_OK ylös ja ylläpitää siirrettävän datan arvoa, kunnes se havaitsee kuittauksen B_OK nousevan reunan. Vastaanottava kelloalue B havaitessaan A_OK signaalin nousevan reunan lukee datan arvon. Kun arvo on luettu, nostetaan kuittaussignaali B_OK ylös siihen asti, kun havaitaan laskeva reuna A_OK signaalista. Näin saadaan data siirrettyä virheettömästi ja varmasti.

Kuvassa 6 esitelty yllä kuvailtu kättelymenetelmä on yksinkertainen ohjelmoida, eikä sitä tehdessä ole välttämätöntä tietää tulevia kellotaajuuksia. Sen vaatima määrä kiikkuja on pieni, mutta ratkaisu luo aina usean kellosyklin viiveen signaaliin. Kättelymetodissa käytetään edellä mainittuun tapaan 2FF-synkronisaattoria estämään metastabilisuuden aiheuttamia häiriöitä kättelysignaalin siirtämiseen. [3]



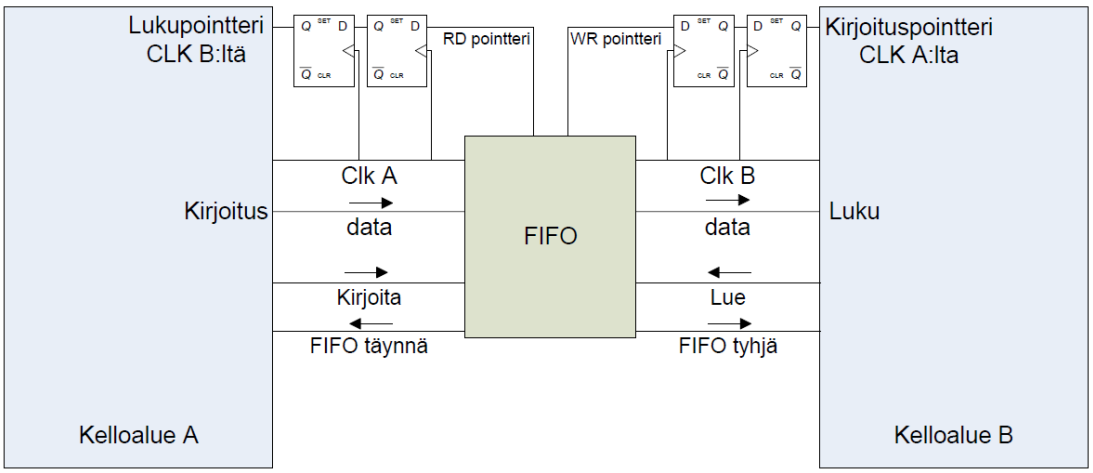

Kuva 6. Kättelymenetelmä.

Kuvassa ei nähdä kättelymenetelmän osana olevaa 2FF-synkronisaattoria kokonaan, vaan lähettävän kelloalueen rekisterit ovat sille merkityn laatikon sisällä. Menetelmässä ei myöskään tarvita 2FF-rakennetta dataväylälle, sillä dataväylän lähettävän pään rekisterillä on paljon aikaa asettua sille annettuun tilaan, kun kättelysignaaleja hallitseva logiikka suorittaa siirtoon vaadittavat toiminnot.

Kättelymenetelmä on myös skaalautuva menetelmä, sillä se voidaan tehdä saman periaatteen mukaisesti riippumatta kättelyn osapuolien, eli kelloalueiden tai signaaleiden määrästä, datan leveydestä tai signaaleiden lukumäärästä. On yleinen käytäntö sitoa iso ryhmä siirrettäviä signaaleja yhteen väylään ja siirtää kaikki yhdellä transaktiolla. Näin säästetään turhia koodin rivejä, joka vuorostaan parantaa kirjoitetun koodin luettavuutta.

### *2.3.3. FIFO*

Sovelluksiin, missä vaaditaan pientä latenssia kelloalueiden välisille datasiirroille, voidaan käyttää kuvassa 7 esiteltyä muistirakennetta. Siinä data kirjoitetaan FIFO-rakenteen sisällä olevaan RAM-muistiin lähdekelloalueesta ja luetaan siltä ulos kohdekelloalueelta. FIFO-rakenteessa etuna ovat pienet latenssit ja datasiirron virheiden vähyys, mutta se vaatii muihin menetelmiin verrattuna piirillä valtavasti pinta-alaa riippumatta siitä, kuinka leveä siirrettävä väylä on. Tästä syystä sitä käytetään vain useamman bitin väylien siirtoon, jolloin se toimii samalla puskurina.



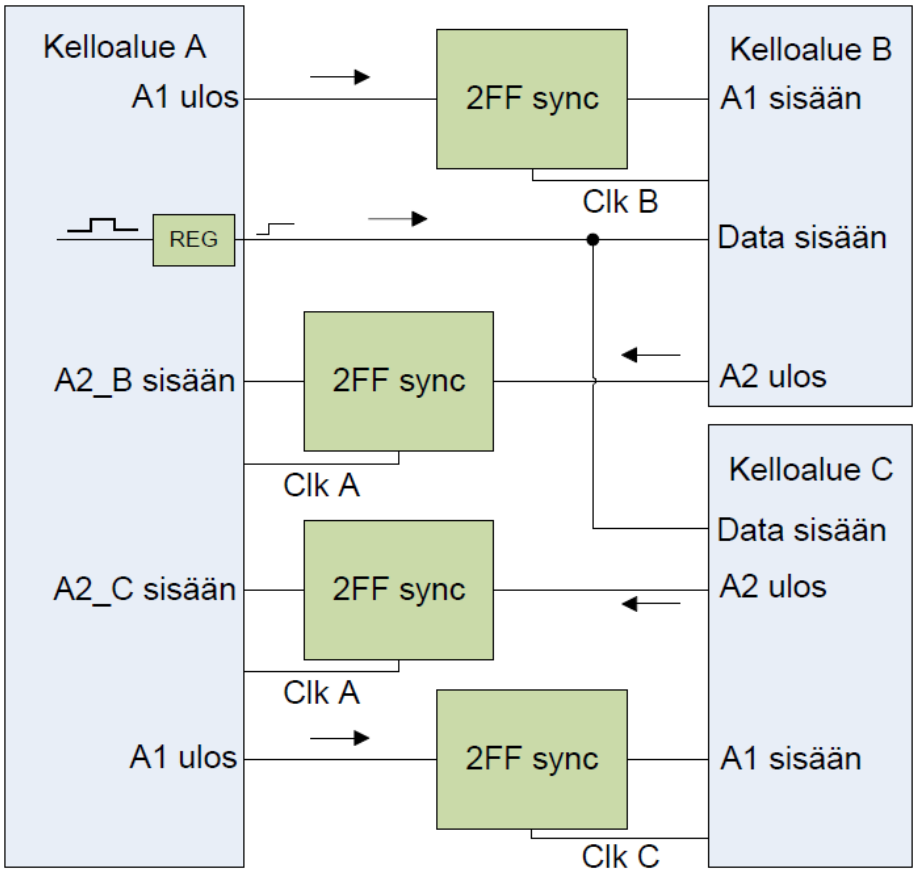

Kuva 7. Yleinen FIFO-sovellus.

Monesti FIFO-rakenteet ovat valmiita synteesityökalun valmistajan luomia osia. Silloin sitä hallitaan yhden bitin levyisillä kirjoita-, ja lue-signaaleilla, jotka ulkopuolinen logiikka luo. IP-moduulin ulkopuolelle tulee vain tieto siitä, onko FIFO täynnä tai tyhjä. Tässä työssä käytetyt näytepuskurit ovat osa itse rakennettua FIFO-rakennetta, jossa luku-, ja kirjoitusosoittajat on tuotu ulos kelloalueiden logiikalle. Näin kirjoitus ja lukusignaalin logiikka kykenevät reagoimaan täyttyvään tai tyhjenevään muistiin ajoissa, jolloin toinen puoli voi hidastaa tai nopeuttaa tarpeen mukaan. Tämä on tärkeää reaaliaikaisissa sovelluksissa, jossa muistin lukemisen ja kirjoittamisen ajoittaminen käyttäjän toimesta tarkasti ei ole realistista, vaan hallintaa täytyy siirtää testipenkkiin. [4]

### *2.3.4. Kolmen tai useamman kelloalueen siirto*

Kun kelloalueita on kolme tai useampi, siirtomenetelmän valinnasta tulee vaikeampaa, sillä pitää selvittää, mikä data pitää siirtää minkä kellorajapinnan yli. Täytyy myös selvittää kaikki mahdolliset signaalit mitkä synkronisoida kaikille kelloalueille. Tässä työssä on pulsseja, joita kaikki kolme kelloaluetta tarvitsevat toimiakseen. Esimerkkinä aloituspulssi, joka voi tulla sisään miltä tahansa kelloalueelta, josta se pitää siirtää kahdelle muulle kelloalueelle. Siirtoa helpottaa se, että pulssin ei tarvitse olla yhtäaikainen kaikilla kelloalueilla. Tämä johtuu siitä, että viive aloituspulssin ja kirjoituksen tai lukemisen välillä pitää olla mahdollisimman pieni ja aina vakio, joka voidaan sitoa johonkin kiinteään, kuten tietyn kelloalueen kellosykleihin.

Kuvassa 8 on kuvaus kättelylogiikan osasta, joka käsittelee kelloalueelta A tulevia pulsseja. Pulssi varastoidaan pitorekisteriin, jota pidetään ylhäällä niin kauan, kunnes kelloalueet B ja C ilmoittavat vastaanottaneensa sen käyttäen A2 signaalejaan. Kun pulssi on siirretty molempiin kelloalueisiin B ja C, voidaan A kelloalueelta laskea A1 sekä pulssin pitorekisteri. Pulssin pituus on kaikissa kelloalueissa yhden kellosyklin verran. Tämä kättelylogiikka on oma prosessinsa, ja sen koko tarkoitus on varmistaa, että pulssi siirretään kaikille kelloalueille.




Kuva 8. Kolmen kellon kättely.

Kuvan 8 metodi on perinteisen kättelymenetelmän tapaan täysin skaalautuva ja se toimii samalla tavalla riippumatta dataväylän leveydestä. Tämäntapaiset n kellon kättelymenetelmät suunnitellaan yksi kelloalue kerrallaan. Ensin ohjelmoidaan ensimmäinen kelloalue siirtämään tarvittavat signaalit muihin n-1 kelloalueeseen, jonka jälkeen tehdään sama seuraavalle ja niin edelleen. Siirrettävien signaalien valinta ja suunnittelu vaikeutuvat mitä enemmän kelloalueita kättelyyn otetaan mukaan. On kuitenkin erittäin harvinaista, että tämäntapaisia ratkaisuja jouduttaisiin tekemään useammalle kuin neljälle kellolle.

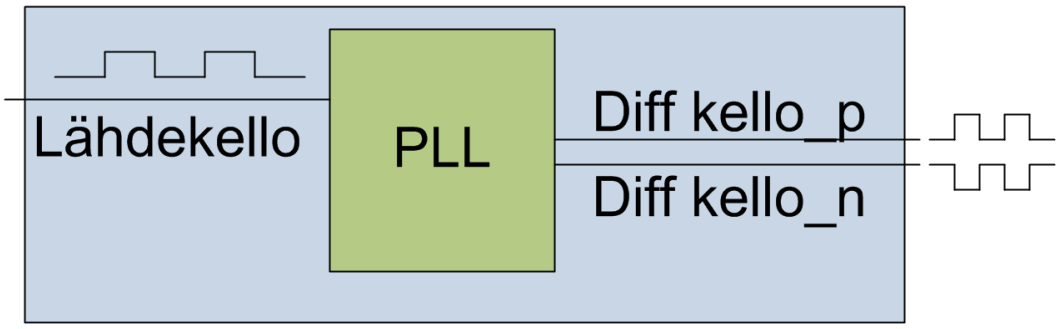
## 2.4. Reaaliaikaisuus

Yksinkertaisin tapa saavuttaa reaaliaikaisuus on käyttää kelloja, jotka toimivat reaaliaikaisella kellotaajuudella, mutta tässä työssä käytetyllä FPGA-levyllä tähän ei päästä, jolloin olemassa oleva logiikka ei sellaisenaan riitä. Tässä työssä tavoitteena on luoda muunneltavalle moduulille reaaliaikainen erikoiskonfiguraatio, joka toimii reaaliaikaisessa prototyypissä.

### *2.4.1. Kellotaajuus*

Reaaliaikaisissa sovelluksissa tarvitaan riittävän nopea kellotaajuus. Tämä tarkoittaa sitä, että saadaan sopiva kellotaajuus, jotta ulkoisesti katsottuna FPGA-levyn toiminta ei poikkea valmiista ASIC-piiristä. Mitä nopeammaksi rajapintojen kellot saadaan, sitä helpompaa on saavuttaa reaaliaikaisuus. Täytyy kuitenkin ottaa huomioon kaikki kellot mitä sovellukseen sisältyy. Tärkeää on määrittää missä kelloalueessa sijaitsee sovelluksen pullonkaula, jonka perusteella voidaan rakentaa testipenkin logiikka ja muut kellot.

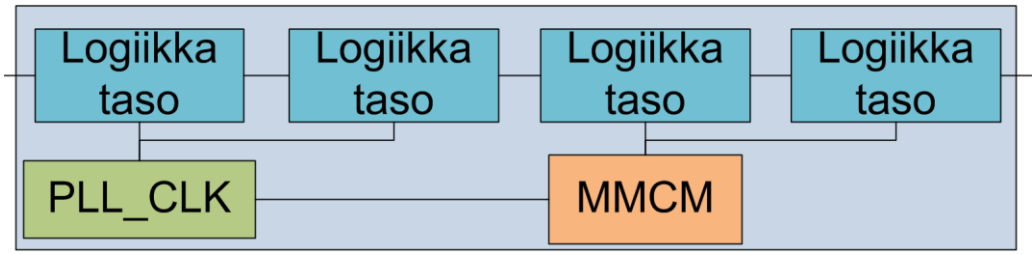
FPGA-levyllä voidaan luoda kelloja käyttämällä levyn omia PLL-rakenteita. Käytetyllä FPGA-levyllä kuvan 9 mukaisesti PLL saa sisääntulonsa joko levyn omalta kellolta tai sille voidaan syöttää kello toisesta sen kahdesta sisääntulotapista. Sisääntulokello on yksipuolinen, mutta PLL-rakenteen kertomalla luomat globaalit kellot ovat sisääntulokellon vaiheeseen lukittuja, differentiaalisia kelloja, jotka omaavat kaksi synkronista puolta, positiivisen ja negatiivisen. Luotuihin kellosignaaleihin voidaan valita valmistajan määrittämien rajojen mukaan haluttu kellotaajuus. Näitä kelloja voidaan käyttää joko yksipuolisina tai differentiaalisina riippuen suunnitelman sietämästä kellosärön määrästä. [6] [7]




Kuva 9. PLL-kellogenerointi.

Nämä PLL-rakenteet eivät ole itse FPGA-piirillä, vaan ne ovat FPGA-levyllä, mihin FPGA on juotettu. Niitä ei myöskään ole aivan loputtomasti, jolloin voidaan päätyä tilanteeseen, milloin PLL-generoidut globaalit kellot eivät enää riitä suunnitelman ja testipenkin yhdistettyihin tarpeisiin. Tämänlaisissa tilanteissa voidaan turvautua prototyyppikohtaisiin kelloihin, joka toimii siihen asti, kunnes prototyyppi tarvitsee enemmän kelloja kuin FPGA-levyllä on PLL-ulostuloja. Näissä tilanteissa voidaan turvautua MMCM-rakenteisiin.

Työn MMCM-rakenteet ovat alustettuja FPGA-piirin valmistajan IP-moduuleita, joilla voidaan kertoa sisääntulokelloa nopeammaksi ulostulokelloksi. Näitä moduuleita voidaan sijoittaa kääreisiin helpottamaan kuvan 10 mukaisesti ajoitusta syvässä logiikassa ja vähentämään kellosäröä. Tämän työn loppuvaiheessa käytetty MMCM-moduuli luodaan kellottamaan toisen IP-moduulin rajapintaa, sillä FPGA-levyn generoidut kellot eivät lukumäärältään riittäneet tähän tarkoitukseen. [8]




Kuva 10. MMCM-moduulin käyttöesimerkki

### *2.4.2. Datanopeus*

Reaaliaikaisuuteen päästään, jos saavutetaan reaaliaikaista toimintaa vastaava datanopeus. Tässä työssä se saavutetaan sisällyttämällä 4 eri näytettä yhteen sanaan. Näytteen ollessa 12 bittinen, saadaan niitä laitettua 4 kpl tunnuksineen yhteen 64 bittiseen sanaan. Datanopeuden ja kellojen puolesta FPGA-testipenkki on pullonkaula, jolloin datanopeuden säilyttämiseksi on käytettävä rinnakkaisia polkuja.

Rinnakkaisten polkujen käyttö ei kuitenkaan riitä, jos testipenkin muistin nopeus ei ole tarpeeksi suuri. Käytössä olevassa FPGA-levyssä on 64-bitin dataleveyden DDR3-muisti, jonka oma systeemikello on 133MHz ja jolle syötetään referenssikello, jonka taajuus on 200MHz. Muistin I/O-kello toimii 400MHz kellotuksella, jolloin suurin mahdollinen transaktionopeus on 800 MT/s. Realistisesti transaktionopeudesta voidaan kokemusten perusteella FPGA-ympäristössä saavuttaa n. 80%, eli tässä työssä 640 MT/s.

Se, riittääkö muistin nopeus käsittelemään reaaliaikaisen sovelluksen tuottamaa datavuota, ei voida tietää varmasti ennen kuin sitä kokeillaan. Ennen aloittamista voidaan tietää vain, onko se mahdollista teoreettisesti.

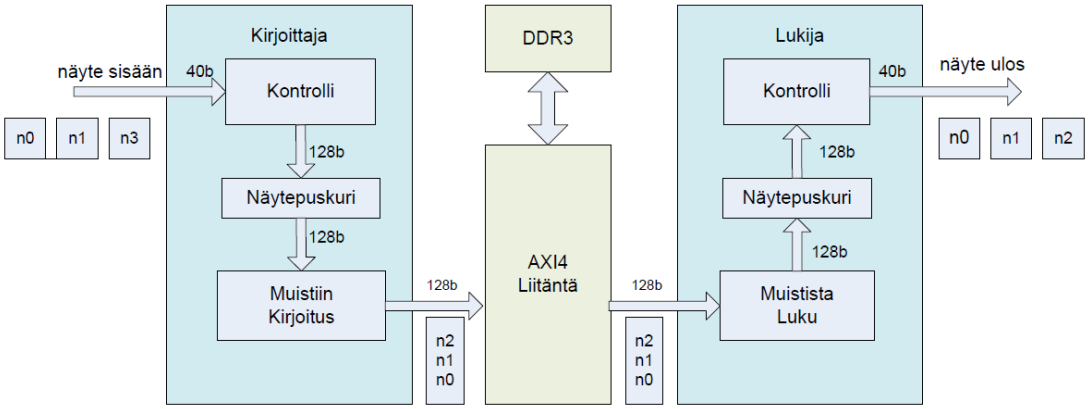
Datanopeuden ja datakaistan käytön optimoinnissa joudutaan käyttämään paljon aikaa. Tällöin voidaan tehdä työn optimointia sillä tavalla, että jos datanopeus muistissa riittää vähiten dataintensiivisellä testitilanteella ilman optimointia, voidaan sitä käyttää todennäköisesti eniten intensiivisellä konfiguraatiolla optimoiden.

# 3. REAALIAIKAISEN OMINAISUUDEN RAKENTAMINEN

Tässä työssä luodaan FPGA-testipenkin kahteen VHDL yleismoduuliin reaaliaikaista käyttöä tukeva erikoiskonfiguraatio. Kyseiset moduulit hallitsevat ja mahdollistavat testipenkin muistiin kirjoittamista ja sieltä lukemista. Tämä on hyödyllistä, sillä näin voidaan RTL-suunnitelmaan syöttää dataa ja kirjoittaa muistiin sieltä tulevaa dataa. Työssä lisätään näihin moduuleihin myös lisäominaisuuksia sopimaan käyttäjien tarpeisiin ja vaatimuksiin.

## 3.1. Moduuleiden muutosvaatimukset

Lähtökohtana on kuvassa 11 esitetty yleismoduulipari, jotka käsittelevät 40-bittisiä näytteitä muistirajapintaan menevälle väylälle, jonka leveys on 128-bittiä. Moduuleille annetaan osoiteavaruudet, hallintasignaalit, ja työn alussa aloituspulssi voidaan antaa vain kirjoittajalle. Tämä moduulipari on vanhaa perua aiemmasta projektista, jolloin se tehtiin yhtä tarkoitusta varten, mutta ne on otettu uusiokäyttöön nykyisessä projektissa. [9]




Kuva 11. Yleismoduuliparin alkuperäinen toiminnallisuus.

Moduuleissa on kaksi kelloa ja ominaisuuksia, jotka toimivat luotettavasti vain yhdellä kellosuhteella. Nämä ominaisuudet ovat tarpeellisia myös muissa konfiguraatioissa ja niiden täytyy toimia tulevaisuudessa kolmella kellolla. Tämän lisäksi kaikki valmiit ominaisuudet toimivat tietyissä testipenkin tilanteissa kyseenalaisesti, jos ollenkaan, jonka takia niiden korjaaminen on myös tehtävälistalla. Korjausvaatimukset sekä uuden kellon olemassaolon huomiointi tulevaisuudessa kuitenkin ovat päällekkäisiä asioita, sillä signaalit täytyy valmistella tuntemattomiin kelloihin, jolloin niiden toiminnallisuus voidaan korjata osana CDC-työtä.

Käsiteltävällä yleismoduuliparilla ei ole työn alkaessa kuin yksi konfiguraatio eikä siinä ole muunneltavuutta. Aiheutunut ongelma koskettaa FPGA-testipenkin ylläpitoa ja kehitystä, sillä kaikki rakenteet joudutaan luomaan yleismoduuliparin puitteissa sen sijaan, että yleismoduulit taipuisivat testipenkin tarpeisiin. Koska FPGA-testipenkki on kirjoitettu Verilog-kielellä, voidaan moduuleihin lisätä portteja ilman vaaraa valmiiden testipenkkirakenteiden muuttumisesta käännöskelvottomaksi.

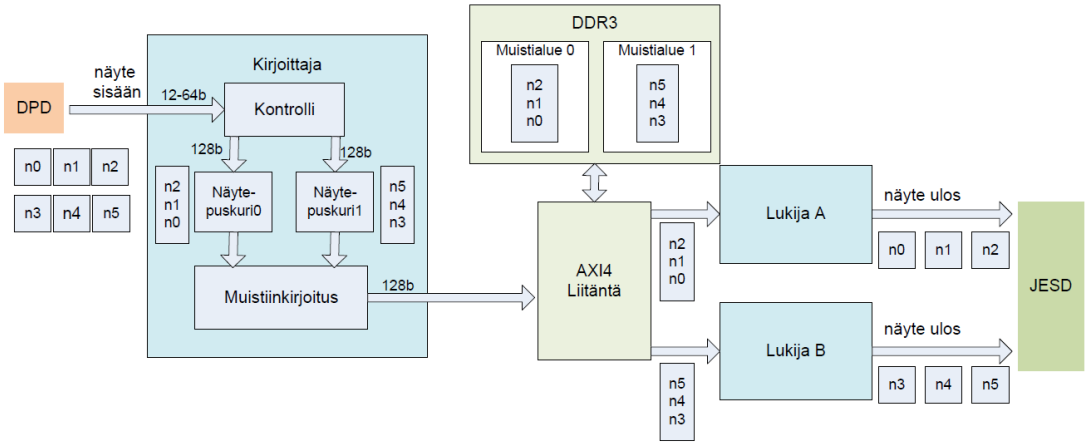
### 3.1.1. Muunneltavuutta tukevat muutokset

Reaaliaikainen sovellus ¼ kellotaajuuteen pystyvällä FPGA-levyllä vaatii useamman jonon pakkaamista yhteen. Tämä vaatii muunneltavan näyteleveyden lisäksi myös muunneltavuutta sille, monta näytettä yhdelle 128-bittiselle väylälle laitetaan. Kirjoittajan sekä lukijan datasiirron aloittamista täytyy myös pystyä hallitsemaan kellosyklin tarkkuudella. On siis lisättävä lukijaan jo kirjoittajassa oleva aloituspulssi ja luoda molemmille vakioviive pulssin lähettämisestä siihen, kun data lähtee liikkeelle.

Lisätään myös testipenkin käytettävyyttä tukeva, vanhemman prototyyppi-insinöörin vaatima kolmas kello, jonka kautta testipenkki pystyy lukemaan moduuleiden statuksen ja hallitsevan niiden toimintaa riippumatta datanopeuksista. Tämän kolmannen kellon on tarkoitus olla testipenkin nopein kello, mutta koska siitä ei ole varmuutta, täytyy valmistautua tilanteeseen, jossa näin ei ole. Moduuleiden on siis toimittava millä tahansa kelloyhdistelmällä, rakennetaan siis kelloalueiden rajapinnat tämä vaatimus huomioiden.

### 3.1.2. Reaaliaikaisuuden vaatimat muutokset

Reaaliaikaisessa sovelluksessa digitaalinen DPD-rajapinta toimii hitaalla kellolla, kun taas JESD204B-rajapinta toimii nopealla kellolla. DPD:ltä tuleva data on lomitettu niin, että yksi TX jono ajaa kahta D/A-muunninta. Kirjoittajan täytyy kirjoittaa kahteen muistialueeseen lähes yhtäaikaisesti vuorotellen muistikirjoituksen resoluution rajoissa. Tästä syystä siihen tarvitaan kaksi yhtäaikaista muistialuetta olemassa olevan yhden sijaan. Lukijoita on yhtä D/A-muunninta varten aina yksi, joten TX-suuntaan niitä pohjustetaan kaksi; Yksi per kirjoittimen muistialue. On siis suunniteltava kuvassa 12 esitelty konfiguraatio.

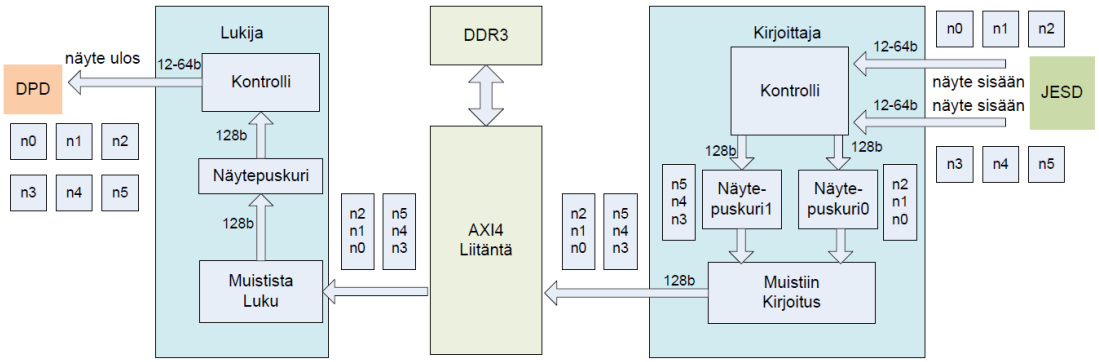



Kuva 12. Esimerkki näytteiden liikkeestä TX-suunnassa.

Kirjoittajaa on käytetty testipenkissä kirjoittamaan dataa muistiin, josta se sitten luetaan testin jälkeen ulos katselmoitavaksi. Tästä syystä siinä ei ole lukijan tapaan osoiteavaruuden vaihtamista kesken kirjoituksen. Osoitteen vaihtaminen on tarpeen, kun kirjoittajan on tarkoitus kirjoittaa muistiin jatkuvan vuon dataa, jolloin se tarvitsisi kaksi muistialuetta per TX-väylä. Osoitteet ovat kuitenkin ohjelmoitu moduuliin hyvin

kankeasti ja niiden muuttaminen vaatisi moduuliin paljon perustusrakenteellisia muutoksia, joka ei olisi työn puitteiden puolesta järkevää. Joten tästä syystä työn määrän optimoimiseksi osoitteen vaihtaminen tehdään moduulin ulkopuolisella testipenkkilogiikalla. Tällä hetkellä kyseisellä ominaisuudella ei ole yleiselle moduuliparille myöskään tarvetta muissa prototyypeissä, eikä testipenkkirekistereissä ole rekistereitä uusille osoitteille.

RX suuntaan tulee takaisinkytkentä, joka kuvan 13 mukaisesti syöttää kahdelta A/D-muuntimelta tulevia näytteitä takaisin DPD:lle. Data lomitetaan samanlaiseen muotoon kuin DPD:ltä tullessaan. Tämä konfiguraatio on siis datan näkökulmasta käänteinen TX-konfiguraatioon verrattuna. Valmiiksi lomitettua dataa ei tarvitse sen DPD-moduulille ystävällisen formaatin vuoksi eritellä eri muistipaikkoihin, helpottaen itse moduulin hallintaa.




Kuva 13. Esimerkki näytteiden RX-suunnasta.

Kuten TX-tilanteessa, tarvitsee RX-kirjoittajakonfiguraatiokin kaksi muistialuetta, koska lukija ei saa lukea dataa alueesta johon kirjoittaja kirjoittaa. Tämä välttää datan mahdollisen korruptoitumisen. Tämänkin konfiguraation osoitteiden vaihto tehdään testipenkkilogiikalla ja osoitteet reaaliaikasovelluksessa tulevat tähänkin kirjoittajaan suoraan lukijan kahdesta muistialueesta.

Reaaliaikaisuus tarvitsee siis tarkkaa data-, ja osoitehallintaa datan formaatin säilyttämisen sekä muistin käytön optimoinnin vuoksi. Työn haasteena tulee olemaan näiden moduuleiden toimintojen sekä viiveiden vakinaistaminen ja ajoittaminen. Valmiissa reaaliaikasovelluksessa testi-insinööri kuitenkin vastaa näistä, jolloin tässä työssä keskitytään varmistamaan, että moduulikonfiguraatioissa on luotu kaikki edellytykset sille, että testaaja voi työnsä tehdä.

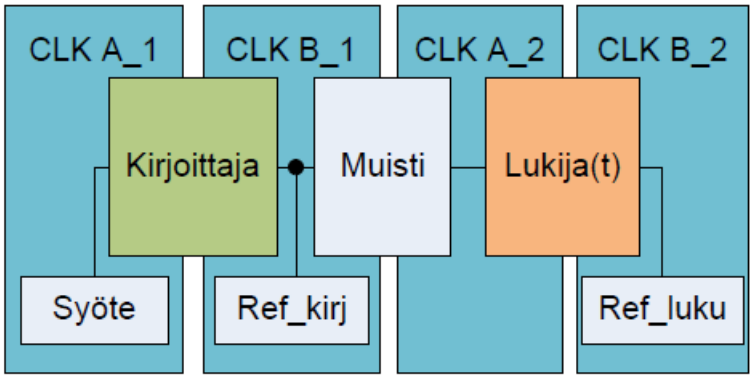
Moduuliparin ohjelmoinnissa voidaan testaajan työtä mahdollistaa laskemalla jokaiseen statussignaaliin sen oma vakioviive signaalin tilan vaihtumisen ja sen vaikutuksen välille, joka on sidottu johonkin moduulin kelloista. On myös pohdittava mahdollisuuksia luoda uusia virhesignaaleja.

## 3.2. Varmennuksen rakentaminen

Tässä työssä välttämättömin on datan liikkeen toimivuus kaikissa konfiguraatioissa. Kun data liikkuu halutusti, voidaan keskittyä kelloihin ja muihin toimintoihin. Simuloidessa voidaan testata tehokkaasti datavuota ohjelmoitavien moduulien läpi ja tarkastella niiden hallinta- ja statussignaaleita.

### *3.2.1. Simulointi*

Yleismoduuleja varten käytössä on vanha VHDL-testipenkki, jossa on muunneltavat kellot sekä tekstitiedostoilla hallittavat syöte ja referenssidata. Datapolkujen sekä kellotuksen muutoksilla saavutettu kuvan 14 mukainen testipenkki soveltuu kaikkien ohjelmoitavien konfiguraatioiden tehokkaaseen testaamiseen. Testipenkissä syötetään dataa kirjoittajaan, joka kirjoittaa sen testipenkin omaan muistiin. Muistin sisältöä verrataan omaan referenssidataan, jolla voidaan havaita kirjoittajan datapolun virheet. Lukijan lähtöä verrataan toiseen referenssiin, josta voidaan nähdä lukijassa tapahtuneet virheet. [10]




Kuva 14. VHDL-testipenkin rakenne.

Testauksen rakenne luo suunnittelujärjestyksen, sillä kirjoittajan datavuo täytyy olla toimiva ennen kuin lukijaa voidaan varmentaa. Kyseinen järjestely on toimiva, koska suurin osa reaaliaikaisuuteen tehtävistä muutoksista on kirjoittajalle.

Testipenkki soveltuu vain datapolun verifiointiin, sillä VHDL-testipenkkien yleinen taipumattomuus pätee myös tässä työssä käytettyyn testipenkkiin. Voidaan siis verifioida vain datapolku sekä sen toimivuus eri kelloyhdistelmillä. Etuna pienellä VHDL-testipenkillä on sen nopea kääntyvyys ja tulosten tuotto, joka pienentää iterointikierroksen viemää aikaa huomattavasti.

Simulaattorilla tehtävässä VHDL-testauksessa toistot ovat hyvin nopeita, sillä jokaiselle testattavalle konfiguraatiolle on oma .do-tiedosto. Tämä simulaattorin konfiguraatiotiedosto sisältää lähdetiedostolistan, muuttujien arvot, syötteet ja referenssitiedostot. Se lukee ja kääntää mukana olevat tiedostot sekä ajaa simulaation loppuun asti yhdellä komennolla, jonka jälkeen se tulostaa testipenkin tulokset simulaattorin teksti-ikkunaan. [11]

### *3.2.2. FPGA-testaus*

Muunneltavalle moduulille tehdään FPGA-levyllä ohjaustesti, eli varmennetaan, että jokainen statussignaali ja kaikki ohjaussignaalit toimivat oikein. Sen lisäksi varmistetaan, että jokainen konfiguraatio voidaan syntetisoida. Varsinainen RTL-testien rakentaminen ei ole osa tätä työtä, vaan työ on onnistunut, kun FPGA-synteesi onnistuu reaaliaikaisilla kellorajoituksilla siten, että muunneltavat moduulit mahtuvat ajoitusrajoitteisiin, sekä testi-insinööri saa käyttöönsä toimivan sovelluksen.

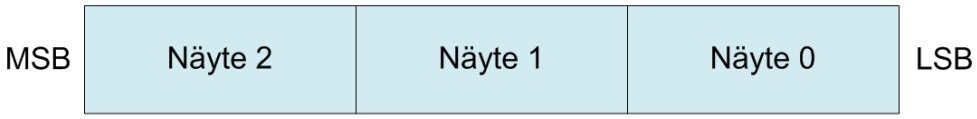
## 3.3. Yleisen muunneltavuuden ohjelmointi

Kuten jo aiemmin työssä on mainittu, on pohjana valmis moduulipari, jonka näyteleveys on 40 bittiä. Moduulit kirjoittavat niille annettujen alku-, ja loppuosoitteiden väliin AXI4 väylän kautta, joko lopettaen kirjoituksen/lukemisen loppuosoitteen jälkeen tai aloittaen uudestaan alkuosoitteesta.

Ennen uusien konfiguraatioiden luomista ohjelmoidaan moduuleiden olemassa oleviin ominaisuuksiin toimivuutta, vakautta sekä muunneltavuutta. Kun moduulit toimivat peruskonfiguraatiossa oikein ja luotettavasti, voidaan siirtyä uusien toimintojen luomiseen. Ohjelmointi jaetaan lohkoihin, jotka eivät vaikuta toisiinsa, jolloin yksi ominaisuusryhmä voidaan tehdä valmiiksi ilman, että siihen tarvitsee tehdä muutoksia siirryttäessä seuraavaan ryhmään.

### *3.3.1. Näyteleveys ja näytteiden lukumäärä*

Näytteiden lukumäärä on sidottu näyteleveyteen, sillä muistiin menevä väylän leveys on 128 bittiä. Kirjoittajan alkuperäinen tapa on laittaa kolmen peräkkäisen kellosyklin aikana kolme näytettä väylään peräkkäin, aloittaen kuvan 15 mukaan alimmasta bitistä. Kun kolme näytettä on asetettu väylälle, kirjoitetaan se muistiin.




Kuva 15. Vanha pakkaustapa.

Lukijalla toiminta on päinvastainen. Siinä 128 bittinen väylä luetaan sisään muistista, jonka jälkeen se luetaan ulos alkaen alimmasta, näyte 0:sta kolmen kellosyklin jonossa. Tämä säilyttää näytteiden järjestyksen siirryttäessä kirjoittajan sisääntulosta lukijan ulostuloon.

Tämä pitää ohjelmoida työn vaatimuksiin paljon taipuvammin, kuitenkaan rikkomatta muuta logiikkaa. Tavoitteiden mukainen toiminta on pohjimmiltaan sama, mutta näytteiden leveys ja niiden määrä täytyy olla vapaasti ohjattavissa parametreilla. Ei ole järkevää tehdä n määrää käyttötilanteita, sillä ohjelmoimalla voidaan saada täysin muuntuva sovellus.

Työssä käytetyssä VHDL kielessä on käytetyllä synteesityökalulla ominaisuus, että muuttuvilla indekseillä syötettäessä dataa väylään on se ohjelmoitava bitti kerrallaan. Tämä vaaditaan, jos halutaan saada aikaiseksi syntetisoituva moduuli. Kuvassa 16 on esimerkki väylään kirjoituksesta. Jokainen näytteen 's_tdata' bitti kirjoitetaan väylään 'dpram_wr_data' samalla kellosyklillä, mutta ne täytyy osoittaa yksitellen koodissa. Kuvassa oleva 'sample_no' muuttuja inkrementoidaan muualla moduulissa, aina näytteen väylään kirjoituksen jälkeen.

```
vector_low                                          := sample_no*sample_bitwidth;
for I in 0 to sample_bitwidth-1 loop
    i_dpram_wr_data(vector_low + I) <= s_tdata(I);
end loop;
```

Kuva 16. Kirjoittajan näytteiden kirjoitustapa.

Lukijassa käytetään myös for-silmukkaa käänteisesti kuvan 17 osoittamalla tavalla.

```
for I in 0 to sample_bitwidth-1 loop
    sample_out(I) <= reg_dpram_rd_data(sample_bitwidth*(sample_no+1)-sample_bitwidth+I);
end loop;
```

Kuva 17. Lukijan näytteiden kirjoitustapa.

Tällä tavalla rakennettu kirjoitus ja luku taipuvat mihin tahansa näytteenleveyteen aina puoleen muistiin liitetyn väylän leveydestä. Tämä rajoitus tulee moduulin käyttämästä näytepuskurista ja yhtälöistä. Näytepuskurissa on kolmen kellojakson viive tulo-, ja lähtöportin välissä. Yhtälön 'sample_no'-muuttuja hallitsee osoitteita ja aiemmin mainitun puskurin datapolkua vaihtamalla tilansa nollaan. Se siis tarvitsee vähintään 2 kellosykliä toiminnallisuuteensa. Koska työssä ei ole tarvetta yli 64-bittiselle näytteenleveydelle, ei voida konkreettisesti perustella tämäntapaisen erikoistapauksen ohjelmointia moduuliin työn aikana.

### *3.3.2. Osoiteavaruuden vaihtaminen kirjoittajassa*

Kappaleen 3.1.2. kuvassa 12 on osoitettu tarve yhden AXI4 väylän käyttämistä kahteen muistialueeseen kirjoittamiseen. Kummastakin näytepuskurista kirjoitetaan sille määrättyyn osoiteavaruuteen. Koska näiden kahden osoiteavaruuden on tarkoitus ajaa yhtäaikaisesti niille kuuluvia lukijoita, on siis niiden kirjoituksen vuorottelu tehtävä mahdollisimman tiheäksi. Työssä käytetään AXI4 protokollassa 256 sanan pituisia purskeita, joten vaihdetaan lähdepuskuri aina 256 sanan välein.

Kirjoittajaan on tarve myös vaihtaa osoiteavaruuksien 0 ja 1 osoitteita kun kehys on kirjoitettu. Tämä olisi mahdollista tehdä lisäämällä 2 uutta aloituspistettä ja lopetuspistettä moduulin portteihin, mutta koska tämän käyttötarkoituksen kehykset ovat yli 10ms mittaisia, uudet osoitteet voidaan asettaa valmiiksi olemassa oleviin osoiteväyliin ennen uuden kehyksen aloittamista. Jotta tämä olisi käyttäjälle helppoa, luodaan moduuliin yksinkertainen rakenne, joka lukee alku-, ja loppuosoitteet vain kerran kehyksen alussa. Näin käyttäjä voi vaihtaa osoiteavaruuden seuraavalle kehykselle heti kun ensimmäisen kirjoitus on aloitettu. Ratkaisu säästää myös

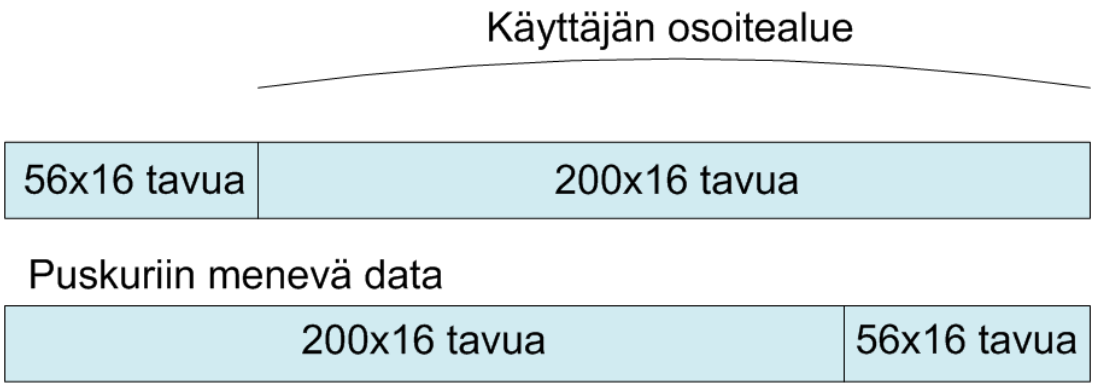
arvokasta tilaa FPGA-levyllä, sillä neljän uuden 64-bittiä leveän osoiterekisterin käyttöönotto testipenkkirekisteriin vaatisi lisää tilaa yleismoduuleille, itse testipenkkirekisterille, sekä johdotukselle.

Tämä ratkaisu on kuitenkin kelpaamaton DPD:n reaaliaika testauksessa, sillä käyttäjä ei pysty vaihtamaan osoitetta tarpeeksi tarkasti. Tässä tilanteessa on myös haasteena, ettei valmiiseen asetusrekisteriin, jolla osoitteet kirjoittajaan asetetaan, ei edellä mainittuun tapaan ole paikkoja tarvittaville osoitteille. Ongelman ollessa kuitenkin tapauskohtainen, voidaan testipenkkiin ohjelmoida kääreen sisään toiminta, joka vaihtaa kirjoittajan osoiteporttiin eri osoitteen, kun kirjoittaja on sen lukenut. Osoiterekisterien puute korjataan käyttämällä kirjoittajien alueiden osoitteina niistä lukevien lukijoiden osoiteavaruuksia. Tämä estää joidenkin testipenkin toimintojen käytön, mutta koska käytetty ratkaisu on käytössä vain yhdessä prototyypissä, on se hyväksyttävää.

### *3.3.3. Osoiteavaruuden resoluutio lukijassa*

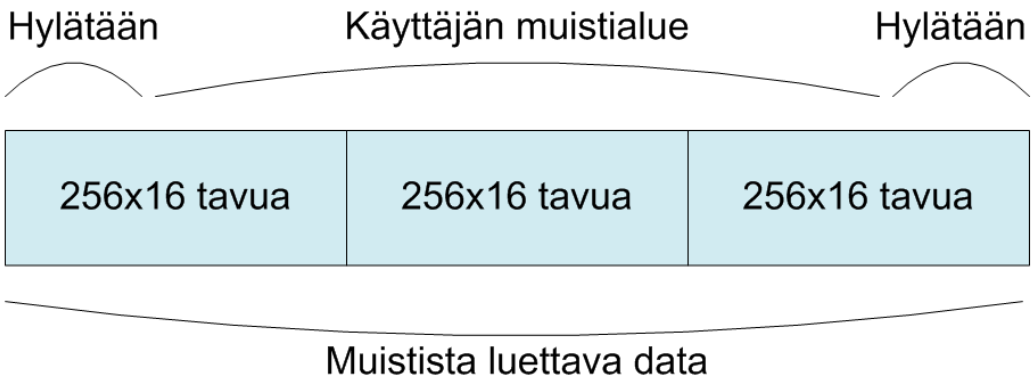
Lukijassa käytetään osoiteavaruuden vaihtoa 1-bittisellä signaalilla, sillä se voi lukea samaa osoiteavaruutta useamman kerran. Lukijat eivät käyttäydy tuhoisasti muistissa olevalle tiedolle, jolloin niiden hallinta voi olla lähempänä ”käynnistä ja unohda” ajattelumallia. Ainoa vaatimus lukijoille on, etteivät ne lue samanaikaisesti muistialuetta, jolle kirjoittaja kirjoittaa.

Useimmiten tätä ominaisuutta käytetään, jos halutaan lähettää muistiin kirjoitetun kehyksen alku, esimerkiksi 46 näytettä, vain kerran, jonka jälkeen luetaan vain sen loppuosaa. Toinen kohde on luettavan muistialueen vaihtaminen vasta, kun seuraava muistialue on kirjoitettu loppuun. Alkuperäisessä moduulissa muistialueen pituuden vaihtelu onnistui vain kappaleessa 3.3.2. mainitussa 256 sanan resoluutiossa. Tämän muuttaminen ei ole kiinteän 256 sanan purskepituuden johdosta mahdollista kirjoittajalle. Lukijassa on kuitenkin mahdollista tehdä ratkaisu, joka toimii ulospäin samalla tavalla kuin muistialueen osan lukematta jättäminen. Kuvassa 18 on esitetty, miten data liikkuu puskuriin, jos luku aloitetaan keskeltä muistialueen resoluutiota ennen resoluution parannuksia. Kuten havaitaan, kyseinen käyttäytyminen johtaa siihen, ettei dataa voida käyttää.




Kuva 18. Muistialueen 256 sanan resoluution aiheuttama ongelma.

Muisti on rakennettu niin, että se on jaoteltu 256x16 tavun laatikoihin. Vain yhdestä laatikosta voidaan lukea yhden purskeen aikana, joten jos aloitetaan laatikon keskeltä, luetaan purskeen loppuun laatikon alkupäästä. Kyseisestä ominaisuudesta johtuen muistin luku on aina aloitettava 4096 tavun monikerrasta. Muistille annettua osoitetta muutetaan aina laatikoittain, ei sanoittain. Tarvitaan kuvan 19 mukainen lukumetodi.




Kuva 19. Lukijan lukurakenne parannetulla resoluutiolla.

Muistista luettavaa dataa ei siis voida rajoittaa, mutta lukijassa sitä voidaan hylätä. Rakentamalla lukijaan 16 tavun resoluution osoiteseuraaja, voidaan lukijan saamasta sanajonosta hylätä kaikki sanat, jotka ovat tulleet muistialueen ulkopuolelta. Tähän vaaditaan myös osoitteen pyöristäjä, sillä vaikka käyttäjä antaa 16 tavun resoluutiolla osoitteet, täytyy muistista lukea 4096 tavun monikerroissa. Jotta dataa, jota halutaan lukea, ei menisi hukkaan, pyöristetään aina aloitusosoite edelliseen ja loppuosoite seuraavaan monikertaan.

### *3.3.4. Kellojen muunneltavuus yhden bitin signaaleilla*

Kellojen muunneltavuus vaatii sitä, että signaalit saadaan siirrettyä kelloalueiden välillä riippumatta kellojen taajuuksista. Joten jos moduulin kelloaluesiirrot tehdään tämän vaatimuksen kanssa, on moduuli aina toimiva. Kellojen muunneltavuudessa on myös otettava huomioon kaikki mahdolliset metastabiilisuudet, eli kelloalueiden välisillä rajapinnoilla täytyy olla tarkkana.

Tulo-, ja lähtökellot ovat tähän asti olleet tiedossa. Tilanne on ollut se, että kellosuhde on aina 1:2 tai 2:1. Kaikissa testipenkissä käytettävissä sovelluksissa asia ei ole näin. On siis syytä käyttää kelloaluesiirroissa kättelyä, sillä se on ainoa CDC-menetelmä, joka on täysin riippumaton kellojen lukumäärästä ja niiden taajuuksista. Moduuleihin lisätään myös kolmas kello, jolloin kättely on kolmen kelloalueen välistä. Kuvan 20 esimerkissä on uuden, kolmannen kelloalueen CA eli kantoaaltosuuntaaja, toisin sanoen aloituspulssi, jota käytetään kelloalueen A logiikassa. Tässä tilanteessa logiikka optimoidaan siten, että kelloalueeseen B ja uuteen kelloalueeseen sys tehdään logiikka, joka siirtää CA signaalin tulosta tiedon varmasti kelloalueeseen A. Signaalia ei tässä tilanteessa tarvitse siirtää kelloalueelle B, koska se ei aja siellä logiikkaa.

```
sys_clk_process : process(sys_clk, rst_n_sys)
begin
if rst_n_sys = '0' then
    CA_sys <= '0';
elsif(rising_edge(sys_clk)) then
    if(CA = '1' and CA_a_ack = '0') then
        CA_sys <= '1';
    elsif (CA = '0' and CA_a_ack = '1') then
        CA_sys <= '0';
    end if;
end if;
end process sys_clk_process;
```

Kuva 20. CA siirto SYS:n näkökulmasta alueelle A.

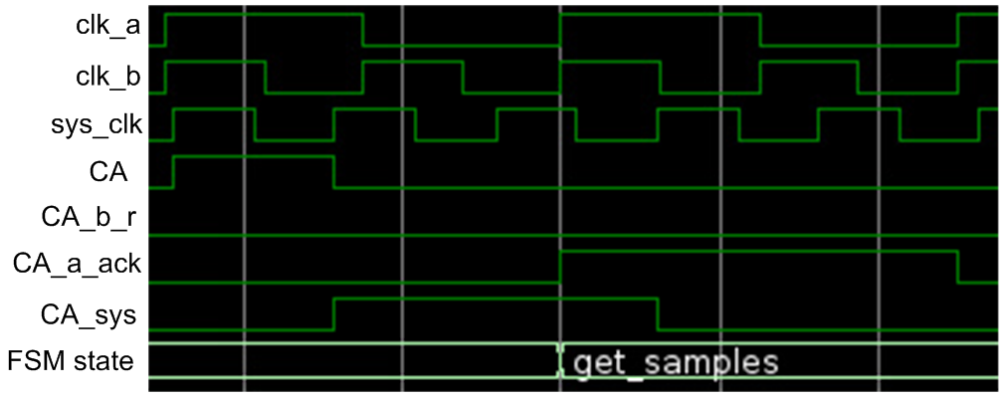
Jokainen kelloalue lukee ’CA’ signaalin sillä oletuksella, että se tulee ko. kelloalueen sisältä. Kättelymenetelmät on rakennettu B ja sys alueisiin, jotta voitaisiin varmistaa, että CA signaali saadaan siirrettyä A:lle. Vastaanottava logiikka on rakennettu tilakone-rakenteen sisään, jonka tila vaihtuu heti kun se havaitsee CA-signaalin. Näin vastaanottavan kelloalueen ei tarvitse luoda pulssimuotoa uudelleen. Kuvassa 21 on kelloalueen A vastaanottava logiikka. Koska kättelysignaalit ovat pitkiä, eikä pulssimuotoa tarvitse säilyttää, voidaan kaikkien kelloalueiden kättelysignaalit asettaa suoraan yhteen if-lauseeseen.

```
-- i_s_tvalid = data valid
if (i_s_tvalid = '1' and (CA = '1' or CA_b_r = '1' or CA_sys = '1') and wr_en_r(1) = '1') then
    sample_no                   <= 1           ; -- sample indexing #(0-number_of_samples_in_wdata)
    main_fsm_state              <= get_samples ; -- next state
```

Kuva 21. Kelloalue A CA-signaalia seuraava if-lause.

Kuvan 22 Simulaattorin aaltonäkymässä sys_clk on nopein kello ja clk_a on hitain. Kuten voidaan havaita, on kelloalue B liian hidas saadakseen kiinni sys_clk alueelta tulevasta CA-signaalista. Sys kelloalue kuitenkin varmistaa, että kelloalue A saa signaalin, joka sitten vaihtaa tilakoneen ’get_samples’ tilaan.

Kuvassa näkyvä CA_a_ack on kelloalueen A kuittaussignaali, joka ilmoittaa muille kelloalueille sen saaneen CA-signaalin. Nähdään siis, että kun kelloalue A havaitsee kahden muun kellon kättelysignaalin laskemisen, laskee se oman kuittauksensa.



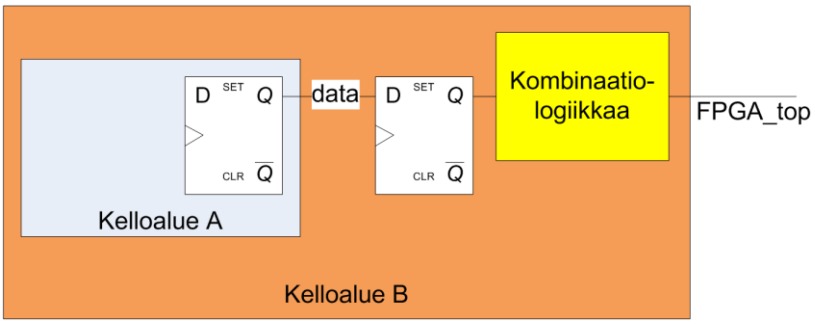

Kuva 22. CA-signaalin siirto kelloalueelle A.

Simulaatiosta siis nähdään, että huolimatta CA-signaalin ollessa kokonaan pois kellon A näkökentästä, saadaan sen välittämä tieto siirrettyä moduulin hitaimmalle kellolle ongelmitta. Mikäli CA tulisi hitaimmasta kelloalueesta, saisivat kaikki kellot siitä kiinni. Kolmen kellon systeemissä siis nopein kello varmistaa, että signaalia tarvitseva kelloalue saa tarvittavan tiedon.

### *3.3.5. Kellojen muunneltavuus dataväylällä*

Dataväylien kellojen muunneltavuudet saadaan toimivaksi käyttämällä FIFO-rakennetta kummankin kirjoittajan sekä lukijan sisällä. Moduuleissa on näytepuskuri, jossa on kello molemmille kelloalueille, väylä ja osoittaja molemmille kirjoitus ja lukupuolille. Kelloalueiden yli siirtyy väylän ulkopuolella vain osoittajat ja muistin kierroslaskureiden lähdöt, sillä lukijan täytyy tietää, missä kirjoittaja menee ja toisinpäin. Näissä kuitenkin on FPGA-levyllä huomattu, ettei näille haluta tehdä viivettä luovaa kättelyä. Osoittajat sekä kierroslaskureiden lähdöt siirretään toiselle kelloalueelle, jotta voidaan seurata mahdollisia yli-, ja alivuotoja. Tähän kuitenkin riittävät yksinkertaiset 2FF-synkronisaattorit tulosta lähtöön ja lähdöstä tuloon. Tämä on perusteltu sillä, että nämä signaalit eivät vaikuta itse moduuleiden datavuohon mitenkään ja selkeyden takia ne halutaan mahdollisimman lähelle virhetilaa.

Yli-, ja alivuotosignaalit luodaan systeemikellon sijasta tulo-, tai lähtökelloalueelle, sillä siten tiedetään varmasti toisen osoittajan olevan aina synkroninen. Tämä vähentää vuodon signaalin viivettä, jolloin se on lähempänä virhetilannetta. Työn moduuleissa signaalit luodaan näytepuskurista lukevaan kelloalueeseen, sillä ensimmäinen vastaantuleva ongelma on usein alivuoto. Toinen syy tähän valintaan on se, että moduuliparin kirjoittajan käyttäminen on ohjelmistopäästä hyvin hallittua, kun taas lukijat voidaan pitää päällä pitkiä aikoja. Virheenä on siis usein liian vähäinen tai hidas muistiin kirjoittaminen.




Kuva 23. Metastabiilisuudelle altis RTL-rakenne.

FPGA-ympäristössä hitaiksi signaaleiksi voidaan kutsua niitä, jotka vaihtuvat yhdestä kolmeen kertaan signaalin tuottavan moduulin operaatiokierroksen aikana. Esimerkki tästä on lukijan FSM-statussignaalit, jotka kertovat missä vaiheessa lukija operaatiossaan on. Näitä signaaleita seurataan silloin, kun moduulin toiminnassa tulee pysähdys. Näissä tapauksissa statussignaali jää yhteen tilaan, josta johtuen kelloaluesiirtoa ei aina välttämättä tarvitsisi tehdä teorian mukaan, vaan sen voi jopa poistaa, tästä esimerkkinä kuvan 23 RTL-suunnittelijan näkökulmasta vaarallinen ratkaisu. Tässä työssä on näihin laitettu aina 2FF synkronisaattori muunneltavuutta ja

eri käyttötilanteiden mahdollistamien metastabiilisuustilojen ehkäisyä tukemaan. Tämä on erikoisuus FPGA-ympäristössä, sillä toisin kuin RTL-suunnitelmaa tehtäessä, FPGA-testipenkissä voidaan tietyissä, hitaissa tapauksissa käyttää 2FF-synkronisointia yli yhden bitin dataleveyksille.

## 3.4. Erikoiskonfiguraatiot TX- ja RX-suuntiin

Reaaliaikaisessa testauksessa käytetään erikoistilanteissa kappaleessa 3.1.2. esiteltyjä konfiguraatioita. Molemmat konfiguraatiot luodaan samoilla yleismoduuleilla, joka vaatii lisää taipuvuutta moduuleilta selkä testipenkiltä. Nämä konfiguraatiot vaativat ohjelmointia pelkästään kirjoittajaan, sillä lukijan osuus näissä voidaan luoda lisäämällä yksi instanssi testipenkissä kuten kuvassa 12. näkyvä lukija B.

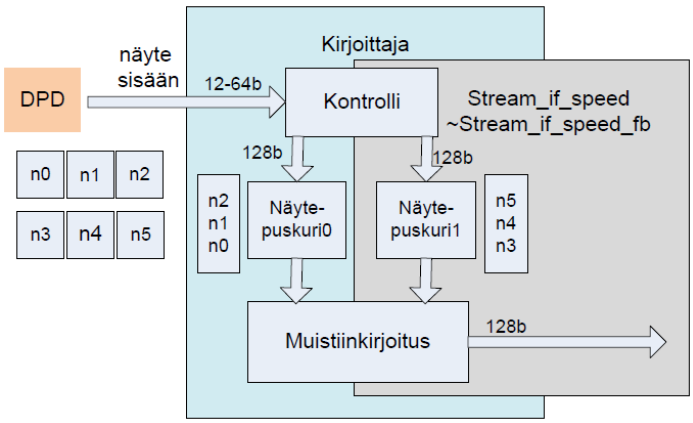
### *3.4.1. Generic-rakenteen käyttäminen*

Generic-rakenne on VHDL vastine parametreille, jotka muuttavat miten koodi käännetään. Työssä on käytössä kaksi tosi-epätosi arvoilla toimivaa muuttujaa, joiden nimet ovat ’stream_if_speed’ ja ’stream_if_speed_fb’. Koska työn konfiguraatiota käytetään aina yhdessä, sidotaan ne molemmat kumpaankin muuttujaan. Käytettyjen muuttujien vaikutus näkyy taulukossa 1. [2]

Taulukko 1. Muuttujien vaikutus

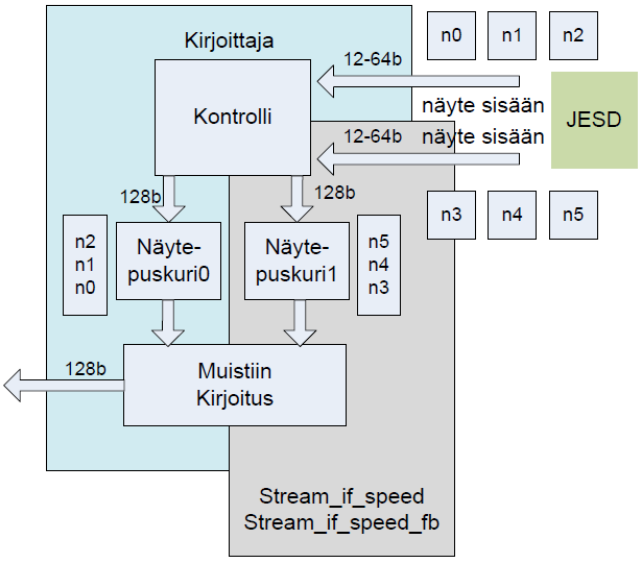
| Stream_if_speed | Stream_if_speed_fb | Valittu konfiguraatio |
|---|---|---|
| epätosi | epätosi | Yleinen |
| tosi | epätosi | Reaaliaika-TX |
| tosi | tosi | Reaaliaika-RX |
| epätosi | tosi | Yleinen |

Viimeistä yhdistelmää ei tässä työssä tarvita. Mahdolliset käyttäjävirheet vältetään, kun jokaisessa rakenteessa, kuten if-lauseissa, käytetään aina joko stream_if_speed tai molempia stream_if_speed ja stream_if_speed_fb. Kuvassa 24 näkyy, mihin osiin nämä kaksi muuttujaa vaikuttavat kirjoittajassa TX-reaaliaikakonfiguraatiossa.




Kuva 24. Generic-muuttujien vaikutusalueet TX-konfiguraatiossa.

Näytepuskurin lisääminen on helppoa, kaikki kontrollilogiikka sekä kirjoitus vaativat kuitenkin tarkkaa ehtolauseiden hallintaa, sillä työssä käytettävät muuttujat ovat mukana kaikissa logiikkalauseissa riippumatta konfiguraatiosta.




Kuva 25. Generic-muuttujien vaikutusalueet RX-konfiguraatiossa.

RX-suunnassa vaaditaan kirjoittajalta portin kuvan 25 mukainen lisäys. VHDL ei kuitenkaan hyväksy porttien lisäämistä tai poistamista muuttujilla, joten yhdessä käytetty portti täytyy olla mukana jokaisessa konfiguraatiossa. Tila-, ja käyttösyistä yleisessä moduulissa ylimääräisten porttien määrä täytyy pitää pienenä.

### *3.4.2. Konfiguraatioiden ohjelmointi*

Uuteen konfiguraatioon vaadittavia rakenteita voidaan tehdä kahdella tavalla. Ensimmäistä, lisäämistä, käytetään, kun ominaisuus on koodin toiminnallisuuden sydämessä. Tästä esimerkkinä on tilakoneen tilojen vaihtamiseen liittyvä logiikka. Työn keskiössä on muiden konfiguraatioiden häirinnän minimoiminen, jolloin lisääminen on helppoa tehdä if-else-lauseiden ehdoilla. Tämän heikkous on kuitenkin koodin luettavuuden heikentyminen nopeasti. Kuvassa 26 on esimerkki luettavuuden romahtamisesta vain kahden konfiguraation lisäämisen jälkeen.

```
if addr_done = '0' then
    i_axi_m_awvalid <= '1' ;
    i_axi_m_wvalid  <= '0' ;
    if (i_axi_m_awvalid = '1' and axi_m_awready = '1' and (stream_if_speed = false or stream_if_speed_fb = true)) then
        addr_done       <= '1'; -- always true at this stage (as not dealing w/ multiple id(s))
        i_axi_m_awvalid <= '0';
        if ( i_axi_m_awaddr < unsigned(ddr3_wr_stop_r2)) then
            i_axi_m_awaddr <= i_axi_m_awaddr + to_unsigned(ddr3_addr_increment,32); -- ==~~  4 * axi_m_awlen
        end if;
    -- stream_if_speed = true stream_if_speed_fb = false use case uses two address ranges on the ddr3
   elsif (i_axi_m_awvalid = '1' and axi_m_awready = '1' and stream_if_speed = true and stream_if_speed_fb = false) then
        addr_done       <= '1'; -- always true at this stage (as not dealing w/ multiple id(s))
        i_axi_m_awvalid <= '0';
        if ( i_axi_m_awaddr < unsigned(ddr3_wr_stop_r2) and sample_memory_toggle = '1') then
            i_axi_m_awaddr <= i_axi_m_awaddr + to_unsigned(ddr3_addr_increment,32); -- ==~~  4 * axi_m_awlen
        elsif ( i_axi_m_awaddr1 < unsigned(ddr3_wr_stop1_r2) and sample_memory_toggle = '0') then
            i_axi_m_awaddr1 <= i_axi_m_awaddr1 + to_unsigned(ddr3_addr_increment,32); -- ==~~  4 * axi_m_awlen
        end if;
   end if;
else -- =~ "if addr done = '1'"
```

Kuva 26. Muistiinkirjoituksen hallintalogiikkaa.

Toisessa if-lauseessa on rakenne, joka mahdollistaa moduulin toimimisen riippumatta käyttäjän syöttämien muuttujien arvoista. Tämänlaiset rakenteet täytyy ottaa huomioon muunneltavien moduulien ohjelmoinnissa. Aloituspiste on siis kaikkien ei-toivottujen toimintojen estäminen sekä väärien syötteiden kestäminen. Vasta sen jälkeen voidaan luoda uusia ominaisuuksia.

Tämäntapaista ”rumaa” koodia joudutaan kirjoittamaan ohjelmoitaessa moduulia, jota ei ole tehty muunneltavaksi. Kun moduuli on alusta asti muunneltava, voitaisiin pohjatyö tehdä tulevaisuuden konfiguraatioiden lisääminen mielessä. Suuret if-lauseet eivät kuitenkaan ole funktionaalisesti heikompia kuin selkeämmät case-rakenteet. Jälkimmäinen on kuitenkin luettavampaa sekä helpommin ylläpidettävää kun mahdollisia loogisia lauseita paljon. Tämänkin työn yksi suurista ohjelmointihaasteista oli luoda kankeasta rakenteesta taipuvaa toiminta säilyttäen.

Toista metodia, jossa luodaan uusia rakenteita, käytetään esimerkiksi, kun luodaan konfiguraatiolle sen omia erikoisominaisuuksia, joita muut eivät käytä. Tämänlainen koodi optimoituu pois, jos konfiguraatio ei ole käytössä. Kuten kuva 27 näyttää, on selkeää ja yksinkertaista luoda konfiguraatiokohtaisia rakenteita.

```
--Burst_toggle controls which sample memory the write burst is sent from.
if (dpram_burst_done_sync = '1' and stream_if_speed = true and stream_if_speed_fb = false) then
       i_dpram_burst_toggle <= not i_dpram_burst_toggle;
elsif (dpram_buffer_read_sync = '1' and stream_if_speed = true and stream_if_speed_fb = true) then
   i_dpram_burst_toggle <= not i_dpram_burst_toggle;
end if;

dpram_payload_sel  <= i_dpram_burst_toggle when stream_if_speed = true else '0';
```

Kuva 27. Käytettävän näytepuskurin kanavointilogiikka.

Tämäntapainen koodi ei häiritse muita rakenteita ja on helposti luettavaa. Se voidaan optimoida käännöksessä pois, eikä se riko moduulia käyttäjän antaessa vääriä arvoja. Muuntelemattoman moduulin tekeminen muunneltavaksi sisältää paljon vanhan koodin muuntelua. On siis oltava tarkkana, ettei rikota vanhoja toimintoja.

### 3.5. VHDL-testipenkkivarmennus

Ohjelmoidut konfiguraatiot ovat datarakenteita muuntavia ja ne lisäävät kaikenlaisten kellosuhteiden käyttöä testipenkissä. Alustava varmennus keskittyy näihin kahteen asiaan, sillä muunneltavassa FPGA-testipenkissä on vaikeaa sitä tehdä. VHDL-testipenkissä siis keskitytään testaamaan taulukossa 2 nähtäviä asioita.

Taulukko 2. Testattavia asioita VHDL-testipenkissä.

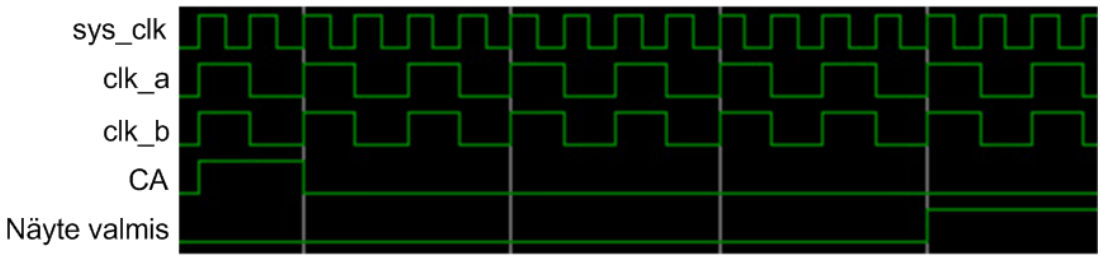
| **Lukija** | **Perus Kirjoittaja** | **TX Kirjoittaja** | **RX Kirjoittaja** |
|---|---|---|---|
| Datapolku | Datapolku | Datapolku | Datapolku |
| Näyteleveydet | Näyteleveydet | Kellosuhteet | Kellosuhteet |
| Aloitusviive eri kelloilla | Aloitusviive eri kelloilla | Kahden Osoiteavaruuden käyttö | Kahden Puskurin käyttö |

Nämä testit rakennetaan valmiiksi ennen ohjelmoinnin aloittamista ja niitä voidaan ajaa simulaattorissa noin minuutissa. Puhdasta kokeiluakin voidaan siis tehdä lyhyen iterointiajan vuoksi. Peruskonfiguraatiolla ajetaan edellä mainittuja datapolkutestejä, joita voidaan ajaa pienillä muutoksilla eri näytemäärillä, näyteleveyksillä, kelloilla, sekä kontrolliajoituksilla. Datapolkutestien tulokset kootaan kuvan 28 mukaisesti heti simulaation valmistuttua.

```
** Note: READ DATAA(39...0) OK: 9216
   Time: 195815 ns  Iteration: 0  Instance: /ddr3_speedbridge_tx_tb
** Note: READ DATAA(39...0) NOT OK: 0
   Time: 195815 ns  Iteration: 0  Instance: /ddr3_speedbridge_tx_tb
** Note: READ DATAB(39...0) OK: 9216
   Time: 195815 ns  Iteration: 0  Instance: /ddr3_speedbridge_tx_tb
** Note: READ DATAB(39...0) NOT OK: 0
```

Kuva 28. Datapolkutestin tietoja lukijoista.

Simulaattori luo myös jokaisesta testipenkin ja moduulin signaalista aaltomuodot, joista voidaan nähdä logiikan toiminnallisuutta sekä digitaalilogiikasta johtuvat viiveet. Se, että dataväylä on testipenkin mukaan kunnossa, ei tarkoita välttämättä sitä, että kaikki toimii halutusti. Jokainen logiikan ominaisuus täytyy siis varmentaa silmin. Kuva 29 on kaappaus aalloista, josta voidaan nähdä lukijan CA – eli lukijan aloituspulssin – jälkeinen viive siihen, että lukijan ulostulossa on oikeellinen muistista luettu näyte. Tämän pohjalta voidaan selvittää, voiko viiveen sitoa minkään kellon sykleihin. Asia katsotaan mahdollisimman monella kelloyhdistelmällä, jotta siitä voidaan olla varmoja. Työn moduulissa CA-näyte valmis-viive on 7 kpl. Clk_b:n syklejä.




Kuva 29. Aaltoikkunan tarkastelua simulaattorissa.

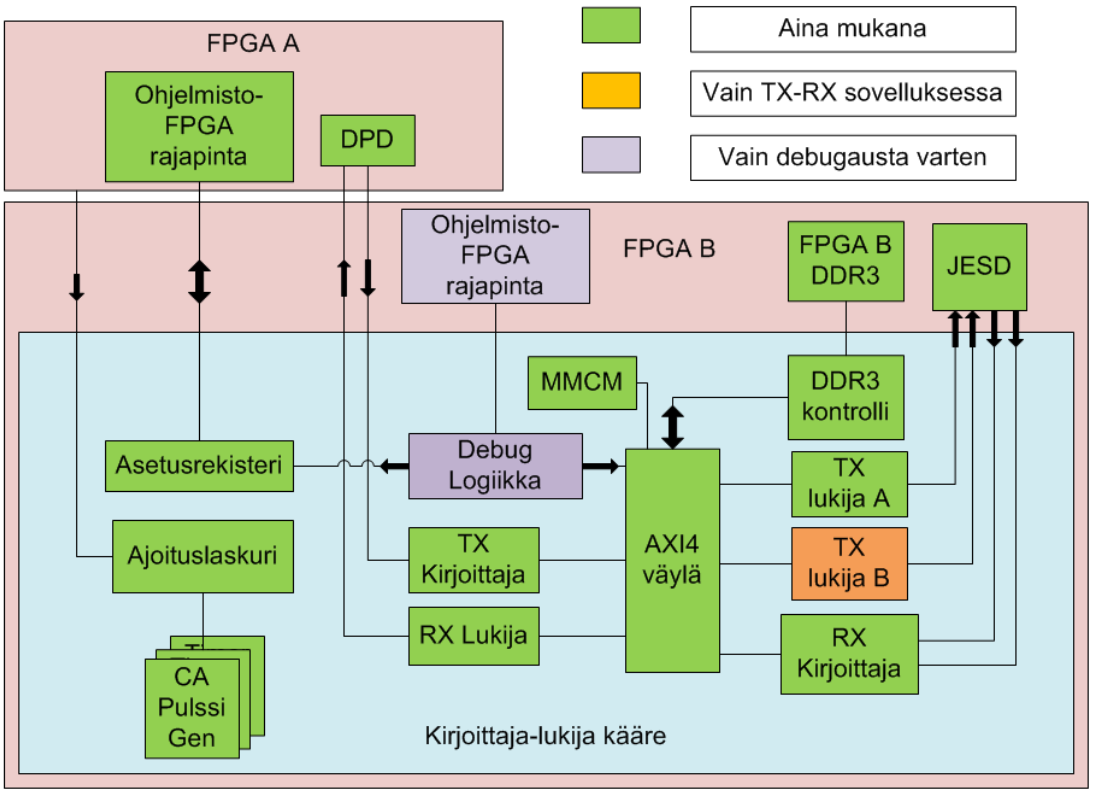
Reaaliaikaisen sovelluksen referenssidatan luominen on työlästä, sillä referenssidata on joka vaiheessa eri. Syötettä tarvitaan kahdenlaista, yksinkertaista, jossa numerot ovat järjestyksessä nollasta 2047 asti, sekä satunnaisdataa sisältävää. Yksinkertaista syötettä käytetään rakennusvaiheessa, jotta simulaattorissa on mahdollista tarkastella kirjoittajan sisäistä vuota ihmisen toimesta. Tätä syötettä pystytään käyttämään myös loppuvarmentamiseen, mutta jotta mahdolliset saman näytepuskurin sanan kahdesti lukemiset saadaan esille, käytetään varmentamiseen myös satunnaisdataa sisältäviä syötteitä ja referenssejä.

VHDL-testipenkissä on todella työlästä varmentaa eri väylässä olevilla näytemäärillä datavuota. Koska kaikki reaaliaikasovelluksessa tehtävät datavuon toiminnat tehdään valmiille näytteistä rakennetulle 128-bittiselle sanalle, voidaan olettaa että testaamalla perustapaus, jossa on 3 kpl 40-bittistä näytettä ja rajatapaus, jossa on 2kpl 64-bittistä näytettä riittää varmentamaan datapolun tämän työn vaatimalla tasolla.

### 3.6. Reaaliaikasovelluksen FPGA-alustan rakentaminen

Synteesin tarkistamiseen käytetyissä ympäristöissä vaaditaan, että konfiguraatiot voidaan alustaa testipenkkiin. Työn tavoitteena on luoda reaaliaikakonfiguraatiot kahden FPGA-levyn järjestelmään. Synteesialustaksi rakennetaan siis tulevaisuudessa täysin käytettävä rakenne, jossa voidaan käyttää joko perus-, tai reaalikonfiguraatioita. Luodaan siis FPGA-testipenkkiin muunneltava kääre, joka toimii ns. yhden linjan sekä työssä rakennetun TX-RX reaalikonfiguraatiossa. Koska ylätasona on muunneltava testipenkki, käytetään mahdollisimman paljon valmiita rakenteita. Uudelleenkäytön kohteena on keskittää varsinkin hallintaa jo olemassa oleviin kontrollereihin. [5]

Koska luotava kääre on täysin uusi, voidaan itse valita ohjelmointikieli. Testipenkin yleinen kieli on verilog, on järkevää luoda kääre sillä. Tiedetään jo, että tämä kääre tulee olemaan eri FPGA-levyllä kuin valmis kontrollilogiikka, joten joudutaan valitsemaan vain ne välttämättömät asiat, joille on levyjen välisessä siirtoväylässä tilaa, loput alustetaan kääreen sisälle. Alla olevassa kuvassa 30 on jäsennelty kääreen sisältyvyydet ja mitä levylle tuodaan ulkopuolelta.



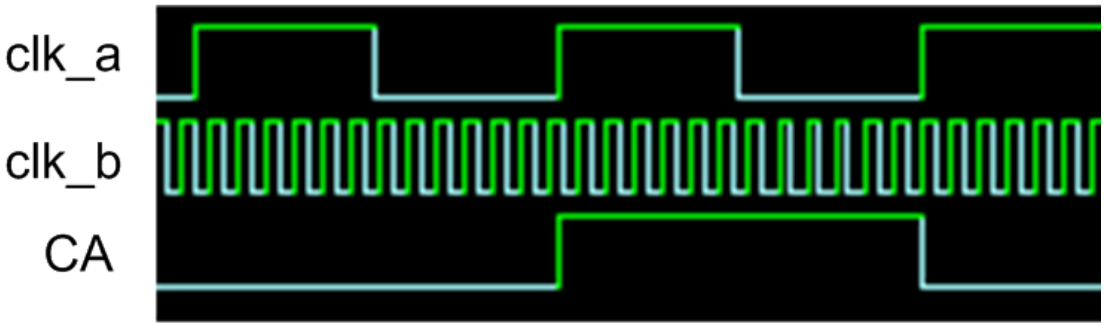

Kuva 30. Moduulit ja tärkeimmät datapolut.

Sisällytetyistä moduuleista AXI4-väylä, virheanalyysi-logiikka ja DDR3-kontrolli ovat Xilinx:in Vivadossa generoituja IP-moduuleja. Ohjelmisto-FPGA rajapinnat ovat generoituja transaktoreita. DPD, JESD ovat testattavia osioita RTL-suunnitelmasta. CA-pulssigeneraattori, kirjoittajat, lukijat sekä ajoituslaskuri ovat Nokian IP-moduuleja. [11]

Kääreessä ei siis tarvitse ohjelmoida mitään funktionaalista, vaan yhdistellään laatikoita ja luodaan taipuvia sekä skaalautuvia rakenteita. Kääre siis kääntyy eri tavalla, jos vaaditaan yhden väylän testejä peruskonfiguraatiolla tai jos halutaan ajaa TX-RX-reaaliaikakonfiguraatiota, sekä silloin, kun virheenkorjaus-rakenteita ei tarvita.

Ohjelmoitu testauskääre alustetaan toisen työntekijän ylläpitämän DPD-reaaliaika ympäristön pohjan päälle. Tämän hyvä puoli on se, että ei tehdä jo tehtyä työtä ja koska iso osa resursseista on valmiiksi sidottu, tiedetään tarkalleen, kuinka paljon voidaan työssä käyttää. Simulaatiotasolla ajetaan rekisteri-, sekä hallintatestit. Pääasiallisesti varmennetaan hallintarekistereiden ja katsotaan, että saadaanko kaikki kirjoittajien ja lukijoiden asetukset ajettua sisään kaikkia mahdollisia teitä pitkin. Jo tässä vaiheessa saadaan siis tehtyä RTL-testaajia hyödyttäviä perustestirakenteita, joiden avulla on helppoa aloittaa testipenkin käyttö. DDR3-muistia ei tässä työssä simuloida, sillä se on vaikeaa sekä hidasta. On havaittu viisaammaksi kääntää rakenne suoraan FPGA-levylle ja testata sitä siellä.

Simulointivaiheessa tarkastellaan VHDL-testipenkissä varmennettuja signaaleja, josta esimerkkinä kuva 31 jossa seurataan CA-signaalin toimivuutta. Tämä on syytä tarkistaa, sillä niitä ei anneta suoraan testistä, vaan ne tulevat testipenkin rakenteista. Tämä simulaatio on hidasta RTL-suunnitelman ollessa mukana, jolloin kovin perustavia ajoja ei kannata tehdä. Tämä johtuu RTL-suunnitelman valtavasta logiikan määrästä, jota simulaattorin täytyy prosessoida. Nopeuseroa näiden kahden välillä kuvaa se, että 30,000 näytteen ajaminen läpi kirjoittajasta sekä lukijasta kestää noin 30 sekuntia VHDL-testipenkillä, kun samassa ajassa FPGA-testipenkin simulaatiossa tehdään 12 rekisterikirjoitusta, 6 jos DDR3 simulointimalli on mukana. Selvä esimerkki siitä, miksei yleismoduulin simulointia tehty RTL-suunnitelman kanssa.




Kuva 31. CA-signaali on saatu nousemaan onnistuneesti.

Kuvasta nähdään jo kellojen suuri ero, joiden kanssa moduulin täytyy toimia. Tässä simuloinnissa kelloissa ei ole vielä fysiikan aiheuttamia viiveitä, joten aitoa toimivuutta ei voida vielä varmentaa. Simulaattorin avulla saadaan myös tarkistettua mukana olevat moduulit käyttämällä sen generoimaa piirikaaviota. Edellä mainitussa näkymässä voidaan myös tarkistella ovatko kaikki tarvittavat signaalit yhdistetty niille kuuluviin paikkoihin.

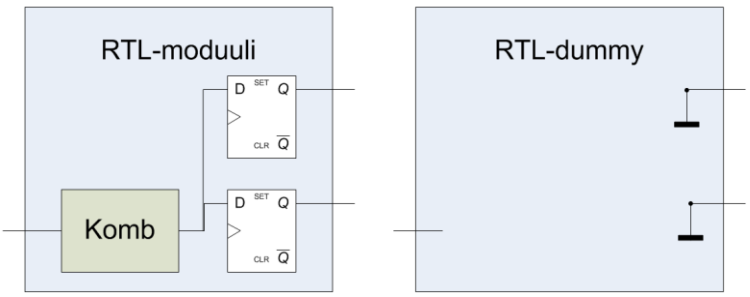
### 3.7. Reaaliaikaympäristön synteesi

Simulointivarmennuksen jälkeinen askel on synteesi. Kuten aiemmin on mainittu, tässä työssä käytetty ympäristö koostuu kahdesta eri mallin ja nopeusluokan FPGA-laudasta. Hitaammassa on kiinni ADC/DAC-kortti, joka on JESD204B:in kautta kiinni työn TX-lukijoissa ja RX-kirjoittajassa. Joudutaan osittamaan edellisen kappaleen kääre mainitulle hitaammalle levylle. Kahden erilaisen levyn ympäristön osituksessa on haasteena niiden erilaiset siirräntäsarjat. Jokainen käytetty I/O-portti täytyy siis erikseen sitoa molemmilla levyillä. Kahden FPGA-laudan yhdistelmässä on myös vaikeutena niiden välillä olevan vapaan väylän rajallisuus. Esimerkiksi kahden FPGA-levyn välinen AXI4 väylä veisi noin 1500 johtoa levyjen väliltä. Asia, joka ratkaistiin vaihtamalla AXI4-protokolla AXI4 liteen, joka vähentää johtojen käyttöä 90%.

Näiden kahden levyn kelloista ainakin yhden täytyy olla synkroninen molemmilla laudoilla, tässä työssä DPD-moduulia ajetaan globaalilla 15,36MHz kellolla, joka generoidaan lautojen PLL-rakenteita käyttäen, jotka sitten synkronisoidaan lautojen välisellä johdolla. Käytetään myös MMCM-moduulia generoimaan nopea kello tästä synkronisoidusta 15,36MHz kellosta. Työssä joudutaan hyödyntämään MMCM-moduulia ajamaan AXI4 yhdistysväylän ja yleismoduulien välistä datasiirtoa, koska synteesissä ei voida käyttää väylän kellottamiseen mitään olemassa olevista kelloista. Tyypillisesti suunnitelman sisällä luotu kello vaatisi kellorajoitteiden asetusta, mutta koska MMCM-moduuli on Vivadossa luotu IP-moduuli, osaa käytetty synteesityökalu siirtää 15,36MHz kellon rajoitteet tälle uudelle kellolle.

Työssä tehdään paljon testipenkin ositusta kahdelle eri FPGA-levylle, joka aiheuttaa rajoitteiden sekä signaalien tappisijoituksien asettamista. Eri nopeusluokan ja eri mallin johdosta kaikki edellä mainituista asioista pitää osata asettaa oikealle FPGA-levylle. Osituksessa joudutaan myös soveltamaan esimerkkisuunnitelmia, sillä esimerkit on luotu toimimaan yhden FPGA-levyn ympäristöillä.

FPGA-synteesissä asetellaan käännettävän suunnitelman kaikki elementit käytettäville levyille sillä tavoin, että ne saadaan johdotettua. Synteesityökalujen vuot kaatuvat, jos ne eivät saa millään tavalla näitä kahta tehtyä. Synteesin lopuksi tarkastetaan ajoitustulokset, josta löytyvät ajoiturajoitteisiin pääsemättömät kohdat, jotka korjataan. Korjaustapoja on muun muassa MMCM-moduuleiden käyttö kriittisten logiikkatasojen välissä, kiikkujen lisääminen kriittisiin datapolkuihin, rajoitteiden löyhentäminen, sekä RTL-suunnitelman keventäminen. Kevennys tapahtuu joko kuvan 32 tapaisilla dummy-malleilla tai muuttamalla RTL-koodia manuaalisesti käyttäen omaa harkintakykyä.



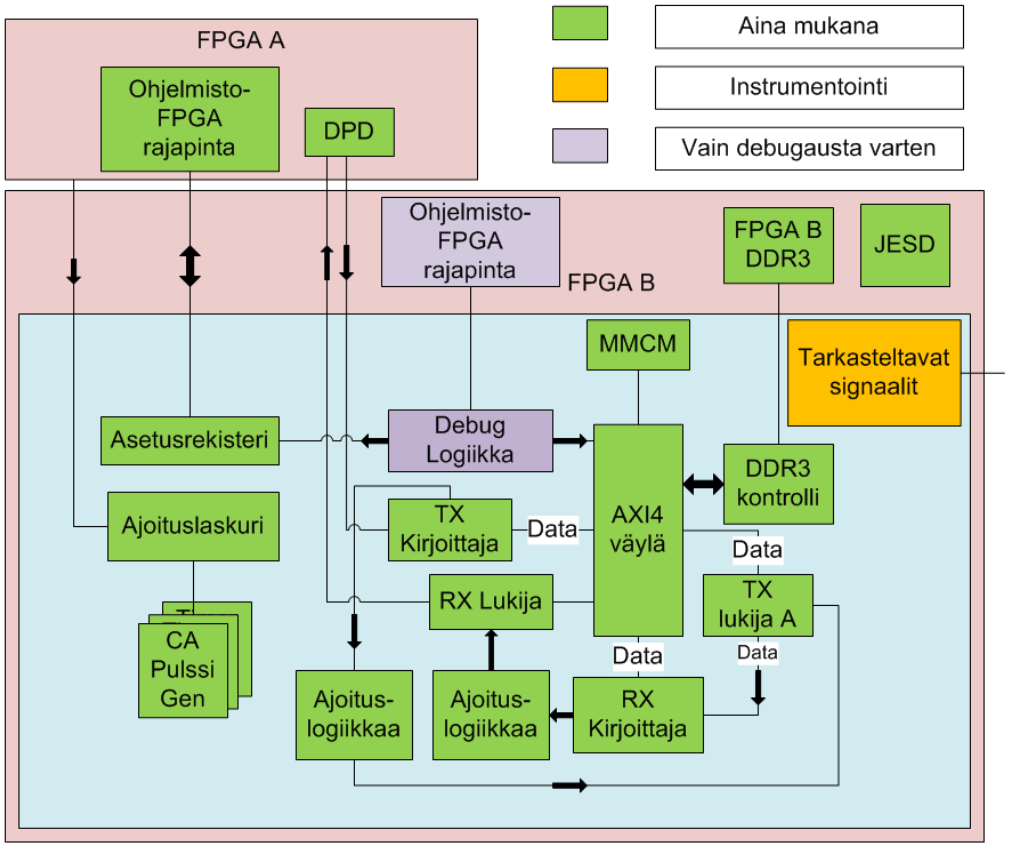

Kuva 32. RTL-moduuli sekä sen dummyversio.

### 3.8. Testiympäristön käyttö RTL-verifioinnissa

Työn aiheena olevaa muunneltavaa testipenkkimoduulia ei voida käyttää ilman, että sille on käyttäjä. Varsinainen RTL-verifiointi tapahtuu usein testi-insinöörin, ei ympäristön rakentajan toimesta. Tarvitaan siis insinöörien välistä kommunikaatiota, jotta työstä saataisiin jotain hyötyä. Tämä on se vaihe suunnitteluvuossa, jossa realisoituu mitä ja miten voidaan realistisesti ympäristöllä tehdä.

Kun moduuli on saatu syntetisoitua levylle siten, että se yltää ajoituksiin, voidaan aloittaa testiympäristön erikoistaminen testaajan tarpeisiin. Työtä käytetään tilanteessa, jossa joudutaan suunnittelemaan yksityiskohdat sen jälkeen, kun pohjarakenteet ovat valmiina, koska ei vielä ole täysin selvää, mitä ja miten halutaan varmentaa. On siis tiedetty, minkälaisia ominaisuuksia tullaan tarvitsemaan, mutta ei ole tiedetty kaikkia yksityiskohtia. Tämänlainen tilanne johtaa siihen, että ennen kuin testaus voi alkaa, joudutaan rakentamaan muunneltavista moduuleista puuttuvia ominaisuuksia kääreen sisälle. Testiympäristö tarvitsee myös sille räätälöidyn käyttötavan yleisille moduuleille, jolloin saadaan muunneltavuudesta paljon hyötyä.

Tässä työssä tehtiin kääreeseen paljon muutoksia yksittäisiin testeihin. Kuvassa 33 näkyy konfiguraatio, jolla saadaan kokeiltua, miten testipenkkiä voidaan käyttää pelkästään DPD-moduulin kanssa. Käytössä on yleismoduulien peruskonfiguraatio 64-bitin näyteleveydellä. Vaikka data kiertää takaisin RX-kirjoittajaan TX-lukijasta, ositetaan JESD FPGA-levylle, koska sitä tullaan tarvitsemaan lopullisessa versiossa.




Kuva 33. Kehityksessä luotu välikonfiguraatio.

Tähän versioon sisältyy instrumentoitavien signaalien, joita testaaja haluaa seurata, valinta sekä reaaliaikaisuuden johdosta tarvittavien ajoitus-, ja hallintalogiikan luominen. Joten yhdellä konfiguraatiolla testaaja saa version, jolla kokeilla ohjelmistoa ja harjoitella kääreen käyttöä, sekä version, jolla kääreen tekijä voi varmentaa sovelluskohtaista koodaustaan.

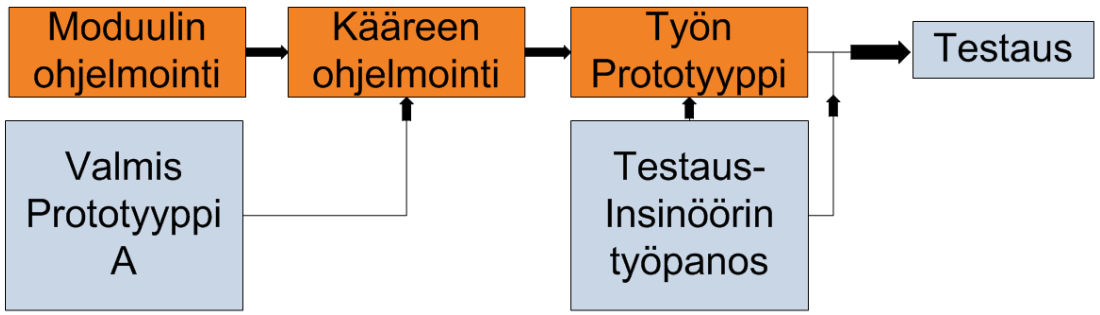
Kuvan 33 konfiguraatio on yksi monista versioista, joita ylläpitäjän täytyy luoda testausympäristön vaatimusten ja testauksen kypsyessä. Näiden versioiden kääntäminen FPGA-ympäristöön kestää kuitenkin kauan, eli prototyypistä voidaan tehdä vain yksi iterointi päivässä. Tämä johtaa siihen, että versioita tehdään noin kaksi-kolme viikossa, sillä testaaja tarvitsee aikaa omassa iterointikierroksessaan.

Kääreen muunnositerointia tehdään niin kauan, kunnes saadaan käyttökelpoinen testiympäristö, jolle reaaliaikasovelluksen tapauksessa on toimivat ajoitusmetodit. Tämä vaiheen pituutta on lähes mahdoton arvioida, sillä lopullinen kääreen muoto on tässä työssä vielä tuntematon testauksen käynnistämisen aikoihin.

Ympäristön kypsyttyä siirtyvät ylläpitäjän tehtävät aktiivisesta ohjelmoinnista tukemaan testi-insinööriä ympäristön käytössä. FPGA-ympäristössä tämä tuki sisältää jonkin verran apua ympäristön käyttämisessä ja RTL-suunnitelman päivittämisessä, mutta suurimmalta osin tuki on virhetilojen analysointia. Analysointiapu koostuu vikatilojen analyysistä ja instrumentoitavien signaalien muuttamisesta ympäristöön. Vaikka testipenkki on verifioitu simuloimalla, voi se toimia odottamattomalla tavalla uusissa testitilanteissa. Tämän tapaisten tilojen ratkaisemiseen tarvitaan ympäristön suunnittelijaa. Syy on siinä, että vaikka testaajalla on syvä ymmärrys itse RTL-suunnitelman toiminnasta, on hänellä harvoin tarkka kuva bittitiedoston testipenkin osista ja sen toiminnoista. Instrumentoinnin lisääminen ja muuttaminen tarvitsevat aina uuden synteesin. Nämä signaalit valitaan siis tarkkaan ja ryhmissä, koska signaaleja ei ole käytännöllistä pitää useassa eri prototyyppiversiossa, jotka ovat instrumentointia lukuun ottamatta identtisiä.

Tämä työn RTL-suunnitelma on jo kypsää, joten työssä ei tarvitse tehdä suurta määrää työtä prototyypin saattamisessa ajoitukseen, sillä työ on tehty jo toisen työntekijän toimesta. Työssä luotu kääre käyttää kahta lukuun ottamatta sille tarkoitettuja kellosignaaleja, jolloin sen ajoitusongelmat ovat yksinkertaisia ratkaista ja seurata. Ajoitusta on kuitenkin tarkasteltava, sillä prototyyppi on merkityksetön, jos sillä ei voida ajoituksen vuoksi varmasti varmentaa siinä olevaa RTL-suunnitelmaa.

Työ on siis saavuttanut tavoitteensa, sillä on tuotettu ajoituksiin yltävä prototyyppi, johon on sisällytetty luotu kääre. Kääreen kaikki vaaditut ominaisuudet toimivat, mutta koska testaus on alkuvaiheessa, ei voida sen toimivuutta sen käyttötilanteessa varmentaa. Tämän työn käyttämät resurssit on näytetty kuvassa 34. Kuvassa on piirretty oranssilla kaikki se, mitä tämän työn aikana luotiin, kun taas harmaalla on kaikki, mitä muiden työntekijöiden tekemästä työstä käytettiin tässä työssä hyväksi.




Kuva 34. Työssä käytettyjen lisäresurssien käyttö.

Kuvasta voidaan nähdä tämä työ aika-avaruudessa, sekä se, mitä oli tavoitteisiin yltämiseen järkevää tehdä. Prototyyppi A on toisen ympäristö-insinöörin tekemä alusta, jonka ylläpidon vastuu yksinkertaisesti siirtyi työtä tehdessä, jolloin vältettiin jälleen päällekkäistä työtä. Tätä prototyyppiä käytettiin reaaliaikasovelluksen kääreen ohjelmoinnissa, simuloinnissa ja lopulta synteesissä. Prototyypin valmistumisen jälkeen alkoi kommunikaatio käyttäjän, eli testausinsinöörin kanssa siitä, mitä prototyyppiin pitäisi vielä sisältää. Kun kaikki prototyyppiin liittyvät ongelmat on saatu ratkaistua, siirtyy työn pääasiallinen vastuu eteenpäin viemisestä testaajalle, jota ympäristön ylläpitäjä tukee ongelmatilanteiden ilmentyessä.

Tämä siis siirtää päävastuun projektin eteenpäin viemisestä eteenpäin seuraavalle erikoistuneelle insinöörille, jolloin ympäristön, eli työn tekijä voi siirtyä muihin tehtäviin. Tulokset reaaliaikatestauksesta, jos se ylipäätään on FPGA-levyllä mahdollista, alkavat näkyä vasta kun testifunktiot ja testit on tehty ja ajettu, kun taas ne eivät voi materialisoitua ennen spesifikaatioiden varmentumista. Tähän työhön otteet kyseisten testien tuloksista eivät kuitenkaan pääse.

# 4. TULOSTEN ARVIOINTI

Työn tulokset olivat tavoitteisiin ja puitteisiin nähden hyvät, sillä alkuperäinen tavoite oli reaaliaikaisen erikoiskonfiguraation luominen sekä muunneltavuuden lisääminen FPGA-testipenkin kahteen yleismoduuliin. Tämä sisälsi näytteiden käsittelyä sekä paljon CDC-ohjelmointia lähes täysin tuntemattomille käyttötilanteille. Työn tavoitteiden ydin, eli simulaatiossa oikein toimivien moduulien synteesi saavutettiin jo hyvissä ajoin, ja työn saattaminen käyttökelpoiseksi alustaksi oli työn loppupään painona. Moduulien toimivuutta peruskonfiguraatiossa on työn aikana varmennettu paljon, sillä ne ovat olleet käytössä lähes koko työn ajan. Moduulit päivitettiin versionhallintaan aina yhden ominaisuusryhmän valmistuttua, vaikka uudet konfiguraatiot olisivat olleet keskeneräisiä.

Erikoiskonfiguraatioita ei tämän työn aikana vielä päästy FPGA-levyllä kokeilemaan niille tarkoitetussa tilanteessa, sillä testaus ei ole siihen valmis. Tehtyä käärettä käytetään yhden TX-linjan ajamisen harjoitteluun. Ei ole viisasta aloittaa kahden D/A-muuntimen kanssa ilman, että yksinkertaisemman yhden linjan käyttö onnistuu. Moduulit kuitenkin simuloitiin, syntetisoitiin, ja ajoitusanalyysit suoritettiin onnistuneesti. Tämä on työn tavoitteisiin nähden tyydyttävä varmennuksen taso.

Työssä olisi voitu luoda kirjoittajaan toinen osoiteavaruus per vuo sekä lisätä siihen lukijassa oleva osoitteen vaihto yhden bitin signaalin avulla. Tätä ei kuitenkaan tehty kolmesta syystä. Ensimmäinen oli ominaisuuden ei-kriittinen tila prioriteettilistalla. Toinen on, että toiminto voitiin luoda suhteellisen yksinkertaisesti itse työn testipenkkikääreeseen. Kolmas on se, ettei kyseistä ominaisuutta koskaan vaadittu moduulilta muuhun kuin DPD-reaaliaikasovellukseen.

Prototypointi on suuri prosessi, jonka yhteen pieneen osaan saatiin näkymä tämän työn toimesta. Prototypoinnissa asia on tehty riittävän hyvin sillä hetkellä, kun se toimii. Työssä on käytetty paljon kokeilu ja erehdys-tapaista työskentelyä, jolloin ei voida sanoa, että on päästy parhaaseen tai edes optimaaliseen tulokseen. Pääasia on pääsy jonkinlaiseen toimivaan ratkaisuun täyttäen työn alussa saadut vaatimukset.

Tuloksia katselmoitiin pääasiassa yhden pulssimuotoisen signaalin muodossa, sillä näin saatiin esiteltyä yhtä asiaa tasaisesti monessa eri tilanteessa. Työssä tämän muotoiset signaalit olivat myös kaikista haastavimpia CDC-ominaisuuksiltaan, joten niiden katselmointi oli insinöörityön puolesta kiintoisinta. Jos tuloksia katselmoidaan useamman signaalityypin ja signaalin pohjalta, levenee työn koko valtavasti ilman mitään varsinaista lisäarvoa itse työlle.

Tuloksia olisi haluttu muun muassa katsella spektrianalysaattorista, jolla tutkikaan ADC/DAC-korttien toimintaa, mutta se ei ollut tämän työn tekoaikana mahdollista. On siis tyydyttävä siihen, että moduulit toimivat simulaatiossa, syntetisoituvat, ja peruskonfiguraatio on toiminut pitkään kaikissa prototyypeissä moitteetta.

# 5. POHDINTA

Tulokset olivat tavoitteiden mukaiset, mutta niiden merkitys jää auki, sillä tämä työ on vain yksi askel koko suunnitteluvuossa. Vuo, joka alkaa RTL-suunnitelman spesifikaatiosta ja loppuu, kun saadaan ASIC-piiriä ajettua prototypoinnin avulla kehitetyllä ohjelmistolla. On siis saatu aikaan tuloksia testiympäristön puolesta, mutta erikoiskonfiguraation tulokset tulevat, jos tulevat. Reaaliaikaista testausta on tämän työn peruskonfiguraatiolla tehty, mutta ei sillä tavalla kuin tässä työssä suunniteltiin.

Parhaimmat ja kattavimmat tulokset, mitä tästä työstä on saatu juuri edellä mainitusta peruskonfiguraatiosta. Tämä johtuu siitä, että sen muunneltavuuteen ohjelmoidut ominaisuudet ovat olleet käytössä prototyypeissä lähes heti niiden valmistumisen jälkeen.

Tulevaisuudessa olisi todennäköisesti tarpeellista ohjelmoida lisää konfiguraatioita ja ominaisuuksia moduuleihin, kuten integroitu kirjoittajan osoitealueen vaihtaminen. Tämän moduulin ohjelmoinnista saadut opetukset ovat tärkeitä, jos tarvitaan uusi muunneltava yleismoduuli, tai nykyinen kirjoittaja-lukijapari päätetään ohjelmoida uudelleen. Nykyisellään moduulipari on ohjelmoitu tasolle, jolla sillä voidaan vielä tehdä testausta seuraavassakin projektista. Kysymys kuitenkin on, että onko tulevaisuudessa tarvetta prototyyppikohtaisesti optimoiduille moduuleille, vai selvitäänkö yleismoduuleilla jotka toimivat aina samalla tavalla? Jos prototyyppikohtaisille moduuleille olisi tarvetta, onko se sen tapainen tarve jota ei voisi mitenkään muuten ratkaista?

Asia on varmasti niin, että moduulien ohjelmoinnin olisi voinut tehdä paremmin, sillä se on kaiken koodin ominaisuus. Työn aikana on tullut hetkiä, jolloin olisi voitu jotain asiaa parantaa, mutta niiden parantelu tai korjaaminen eivät olisi näkyneet millään tavalla nykyisessä prototypoinnissa, joten ne menivät prioriteettilistan pohjalle, paikkaan, josta harva asia koskaan esille tulee.

Työ saatettiin loppuun toivolla, että sen avulla voitaisiin rakentaa käyttökelpoinen ympäristö. Tämä on toivon tasolla juuri siitä syystä, ettei voida olla varmoja, että tavoiteltu asia toimii ennen kuin sitä on kokeiltu. On siis päästy kohtaan missä työn jatkamisen vastuu siirtyy seuraavasta vaiheesta vastaavalle insinöörille.

# 6. YHTEENVETO

Työn tavoitteena oli luoda reaaliaikainen sovellus moduuliin, jota käytettään useassa eri FPGA-prototyypin testipenkissä. Tämä täytyi saavuttaa siten, että moduulin muuntelu ei vaikuta negatiivisesti muihin moduulin käyttötilanteisiin. Lopullinen sovellus siis tehtiin huomioiden muut, jo valmiit sovellukset.

Muunneltavan moduulin haasteet, kuten vielä tuntemattomat kellotaajuudet, metastabiilit tilat ja niiden vaikutusta neutralisoivat CDC-metodit ovat tämänlaisen ohjelmoinnin keskiössä. Työssä käytiin läpi näiden haasteiden aiheuttamat toiminnot sekä työn askeleet, joilla ongelmat ratkaistiin.

Esillä oli erilaisten varmennusalustojen, kuten simulaattorin, heikkouksia sekä vahvuuksia. Päätökset varmentamisen työkalukohtaisesta laajuudesta perustellaan. Näiden alustojen rakenteita sekä sijaa ASIC-suunnitteluvuossa sivuttiin, jolloin nähtiin mikä prototypoinnin rooli on.

Tarkemmin työssä käytiin läpi testipenkkimoduulin uudelleenohjelmointia muunneltavaksi ilman, että pohjasovellusta muutettiin. Nähtiin miten spesifikaatio voi muuttua kesken työn, sekä miten rajataan ja huomioidaan moduulin vaatimukset. Hyväksi havaittu ohjelmoinnin lähestymistapa prototypoinnissa näytti, miten voidaan helpottaa suunnittelijan sekä käyttäjän työtä minimaalisella vaivalla.

Työssä tiivistettiin, miten generic-rakenteilla voidaan VHDL-kielessä luoda moduuli, joka on helposti käyttäjän muunneltavissa. Havainnot hyvistä sekä huonoista käytännöistä ohjelmoidessa tulivat ilmi. Esitettynä oli myös moduulin varmennus ennen FPGA-levylle syntetisoimista.

Synteesivaiheessa tarkasteltiin simulaatiossa varmennetun suunnitelman kääntämistä FPGA-levylle. Tähän liittyi tapoja keventää RTL-suunnitelmaa sekä muita resursseja, joiden avulla voidaan ratkaista synteesissä vastaantulevia ongelmia ja saada ympäristö ajoituksiin.

Lopulta esiteltiin muunneltavan moduulin osaa reaaliaikaisessa prototyypissä ja miten ympäristöä voidaan muokata vastaamaan sitä käyttävän testaajan tarpeita. Tähän liittyen tuotiin esille asioita, jotka vaikeuttavat ympäristön kehittämistä sekä ylläpitoa.

Vaikka lopullista käyttökohdetta ei työssä nähty, saatiin rakennettua sen FPGA-ympäristö tilaan, jolla voidaan tutkia sitä, onko lopullinen tavoite mahdollinen työssä käytetyllä ympäristöllä. Riippumatta ympäristön suorituskyvystä, on se luonut pohjan samantapaisille sovelluksille tulevaisuudessa.

Koska työssä tehtiin asioita, joita ei ole aiemmin tehty, on sillä epäonnistuessaankin merkitystä prototypoinnin FPGA-ympäristöjen mahdollisuuksien kartoittamisessa.

# 7. LÄHTEET

[1] IEEE Computer Society. (2006) IEEE Standard for Verilog Hardware Description Language, 590p. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1620780

[2] IEEE Computer Society. (2008) IEEE Standard VHDL Language Reference Manual, 626p. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4772740

[3] Tieto- ja sähkötekniikan tiedekunta. (2017) Digitaalitekniikka 2. Oulun yliopisto, 339p.

[4] Tieto- ja sähkötekniikan tiedekunta. (2017) Digitaalitekniikka 3. Oulun yliopisto, 206p.

[5] Synopsys. (2017) HAPS Prototyping User Guide, 722p. Saatavissa: https://solvnet.synopsys.com/dow_retrieve/latest/protocompiler/ni/protocompiler_user_guide.pdf

[6] Synopsys (2017) HAPS-80 S26 Reference Manual. Saatavissa: https://www.synopsys.com/apps/protected/hapssupportnet/download.php?file=cd/manuals/doc-00000022_haps-80_s26.pdf

[7] Synopsys (2017) HAPS-DX7 S4 Reference Manual. Saatavissa: https://www.synopsys.com/apps/protected/hapssupportnet/download.php?file=cd/manuals/sh100000329_haps-dx7_s4_ref_man.pdf

[8] Xilinx (2017) 7-series FPGA Datasheet: Overview. USA. Saatavissa: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[9] Juntunen V. (2015) AXI4-Stream based test data insertion module for an FPGA-based prototyping environment. Nokia Networks, 44p.

[10] Mentor Graphics. (2016) Questa SIM User's Manual, 1928p. Saatavissa: https://optima.oulu.fi/learning/id76/bin/doc_show?id=299195

[11] Xilinx. (2016) Vivado Design Suite User Guide, 288p. Saatavissa: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug901-vivado-synthesis.pdf