Tampere University

Thomas Mikkonen

CONTINUOUS INTEGRATION IN SYSTEM-ON-CHIP DESIGN

Faculty of Information Technology and Communication
Bachelor Thesis
May 2020

ABSTRACT

Thomas Mikkonen: Continuous integration in system-on-chip design
Bachelor Thesis
Tampere University
Degree Programme in Computer Sciences
May 2020

Continuous integration is a development practice that aims to make software development faster while keeping code functional. In continuous integration build and tests are run automatically and the developer gets fast feedback from his commitments. The developer also must commit code often so the functionality of code will be checked regularly.

The point of this thesis is to find out if continuous integration can be used in the system-on-chip design. The possibility of using continuous integration in the system-on-chip design could give similar benefits to system-on-chip design as it has given in the software development. This thesis will be using a literature review as a method to find out the possibility of continuous integration usage in the system-on-chip design.

First, this thesis presents important concepts for the topic: system-on-chip and two types of system-on-chip: field-programmable gate array and application-specific integrated circuit, continuous integration, and register-transfer level. After that thesis focuses on explaining what continuous integration is on a more profound level and how it is used in software development. In the end, this thesis tries to answer to the research question, that is it possible to use continuous integration in the system-on-chip design, by comparing continuous integration in software development and in system-on-chip design.

As a result of the comparison, this thesis states that it is possible to use continuous integration for the system-on-chip design if some modifications are made to continuous integration usage. The modifications are essential because of the differences in system-on-chip design and software development. Those modifications needed are mostly related to having multiple teams in the system-on-chip design and also related to different development times between system-on-chip design and software development.

Keywords: continuous integration, system-on-chip, system-on-chip design, software development

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Thomas Mikkonen: Jatkuva integrointi järjestelmän sirun suunnittelussa
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2020

Jatkuva integrointi on kehitystapa, jonka tavoitteena on tehdä ohjelmistokehityksestä nopeampaa ja samalla pitää koodin toimivana. Jatkuvassa integroinnissa koodin kääntäminen ja testaaminen tehdään automaattisesti, kehittäjä saa myös nopeaa palautetta koodistaan palautuksen yhteydessä. Kehittäjän pitää tehdä palautuksia usein jotta koodin toimivuus tulee tarkastettua säännöllisesti.

Tämän opinnäytetyön tarkoituksena on selvittää, voidaanko jatkuvaa integrointia käyttää järjestelmän sirun suunnittelussa. Jatkuvan integroinnin käyttö voisi antaa samanlaisia hyötyjä järjestelmän sirun suunnittelulle kuin mitä se on jo antanut ohjelmistokehitykselle. Tässä opinnäytetyössä käytetään menetelmänä kirjallisuuskatsausta, jonka avulla selvitetään voiko jatkuvaa integrointia käyttää järjestelmän sirun suunnittelussa.

Ensimmäisenä tämä opinnäytetyö esittelee tutkielman aiheen kannalta tarpeelliset käsitteet: järjestelmän siru ja kaksi eri tyyppiä järjestelmän sirusta: kenttäohjelmoitavan porttiryhmän ja sovelluskohtaisen integroidun piirin, jatkuva integrointi ja rekisterisiirtotaso. Tämän jälkeen tutkielma keskittyy selittämään syvällisemmin, mitä jatkuva integrointi on ja miten sitä käytetään tällä hetkellä ohjelmistokehityksessä. Lopuksi tämä työ pyrkii vastaamaan tutkimuskysymykseen, voiko jatkuvaa integrointia käyttää järjestelmän sirun suunnitteluun. Tuloksen saadakseen tutkielma vertailee miten jatkuvaa integrointia käytetään ohjelmistokehityksessä ja miten sitä voitaisiin hyödyntää järjestelmän sirun suunnittelussa.

Vertailun perusteella tämä tutkielma esittää, että jatkuvaa integrointia on mahdollista käyttää järjestelmän sirun suunnittelussa, jos jatkuvan integroinnin käyttöön tehdään joitakin muutoksia verrattuna ohjelmistokehityksessä käytettyyn jatkuvaan integrointiin. Muutokset ovat välttämättömiä, koska järjestelmien sirun suunnittelu ja ohjelmistokehitys eroavat toisistaan monilla eri tavoilla. Nämä tarvittavat muutokset liittyvät useimmiten siihen, että järjestelmän sirun suunnittelussa on useita työryhmiä, ja myöskin siihen, että järjestelmän sirun suunnittelulla ja ohjelmistokehityksellä on eri mittaiset kehitysajat.

Avainsanat: jatkuva integrointi, järjestelmän siru, järjestelmän sirun suunnittelu, ohjelmistokehitys

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

TABLE OF CONTENTS

1 Introduction

Continuous integration (CI) is a way how the software should be done. In continuous integration, the build and testing of the code are automatized for fast feedback. The build and testing happen always when a developer commits code, the commits should be frequently. (Fowler, 2006)

In software development, continuous integration has been used for a while now and it has been giving some benefits for development. The point of view of continuous integration is removing manual labor and verifying old functionalities (Duvall, Glover, and Matyas, 2017). With continuous integration, it is possible to get rapid feedback from the development process by automatic testing. Because of the rapid feedback, errors are found out earlier and fixed faster than without continuous integration. (Duvall, Glover, and Matyas, 2017)

Software teams that use continuous integration have noticed continuous integration to be beneficial for development. (Fowler, 2006) Continuous integration is now used also in the system-on-chip design by some companies, however, continuous integration should be investigated more to get similar benefits as in software development.

This thesis is going to investigate continuous integration in the system-on-chip design. Two system-on-chip types, a field-programmable gate array (FPGA) and an application-specific integrated circuit (ASIC), will be briefly introduced however, the focus of this thesis will be on overall system-on-chip design. The goal of this thesis is to find out can continuous integration be used in the system-on-chip design.

This thesis is going to start by first explaining the concepts that are necessary to understand in system-on-chip design. After concepts thesis presents some background information about continuous integration in chapter two. After the background has been handled this thesis is moving on to the continuous integration in the hardware. Chapter three first compares continuous integration in hardware and software then discusses the problems of continuous integration in the system-on-chip design and finally discusses the usage of continuous integration in the system-on-chip design.

2 Background

The next chapter will define the concepts used in this thesis. First, it defines a system-on-chip and after that two system-on-chip types: a field-programmable gate array and an application-specific integrated circuit. After the system-on-chip definitions thesis moves on defining a register-transfer level description. After all the definitions, continuous integration and the main parts of the continuous integration system are set out, and also usage of continuous integration in software is presented.

2.1 System-on-chip

A System-on-chip (SoC) architecture is an application domain that has tailored processors, memories, and interconnections on one chip. The difference between system-on-chip and the general-purpose computer is that when developing the system-on-chip device, the application that the device is going to run is known therefore all the parts of the system can be chosen for a specific use.

System-on-chip design has multiple different types. The types are categorized by their programmability. Usually, when the programmability of a system-on-chip rises the performance drops. (Flynn & Luk, 2011) In this thesis, only a field-programmable gate array and an application-specific integrated circuit are introduced. The SoC types were chosen for this thesis because they were enough different from each other to give an understanding of a variety of SoC types. In this thesis, system-on-chip and hardware are synonyms.

2.1.1 Field-programmable gate array

A Field-programmable gate array (FPGA) is an SoC type that can be programmed to the needed application without the need of making a specific device for the application. Because of programmability, the multiple FPGAs can be produced for different applications without the need of changing its parts which make production faster and cheaper than application-specific integrated circuits. (Flynn & Luk, 2011)

On technical basic, an FPGA is a matrix of configurable logic blocks that are linked to each other by a completely reprogrammable interconnection network. In FPGA the connections and the logic blocks are controlled by the memory cells so the component can achieve all required application specifications. (Monmasson & Cirstea, 2007)

2.1.2 Application-specific integrated circuit

An application-specific integrated circuit (ASIC) is an integrated circuit that is specially designed for one specific usage and probably to only one customer (Zhang, 2010). Because ASIC is designed for only one application it can give out higher performance, but the cost and flexibility suffer (Flynn & Luk, 2011). An ASIC can be also categorized into different versions of ASIC, for example, a full custom, a semi-custom, and a structured ASIC. Different versions tell the possibility to modify the circuits (Zhang, 2010). Even though some versions of ASIC can be modified they are still done for one purpose and do not have the flexibility of an FPGA.

2.2 Register-transfer Level

The register-transfer level (RTL) description is an abstraction of a circuit. The RTL abstraction of the circuit has simple gates that form modules. Modules include data routing components, functional units, and storage components. (Chu, 2006)

The abstraction level of the RTL is lower than the behavioral description, but the RTL description has also a higher level of abstraction than the netlist description. The RTL design has a detailed relation of data flow and clock. (Churiwala & Garg, 2011).

2.3 Continuous integration

Continuous integration (CI) is a practice regarding how software development should be done. In CI, building and testing your code is done automatically and the objective is to get feedback for errors in builds and tests as soon as possible. This has been noticed to make software development faster by software teams. (Fowler, 2006) Important parts of continuous integration are pipelines, version control repository, and continuous integration server.

In continuous integration, pipelines are used to get fast feedback from success (Olausson & Ehn, 2015). In the pipeline, the different tests and procedures are split into individual stages. If an error occurs in one stage it is reported immediately without the need of going the whole system through. (Olausson & Ehn, 2015)

A version control repository is necessary for getting continuous integration to work. The idea of a version control repository is to help you to manage changes done to source code and documentation. All the work is saved to the one version control repository that you can use for selecting a different version of the project. (Duvall, Glover, and Matyas, 2017) In case you break your code, you can get a working version from the version control repository.

A continuous integration server is a server that checks for changes in a version control repository frequently, usually once in a couple of minutes (Duvall, Glover, and Matyas, 2017). If CI server notice changes in the repository, then it retrieves the files and builds the script. For reporting the results of builds CI server commonly uses a dashboard that shows information of build. (Duvall, Glover, and Matyas, 2017)

2.4 Continuous Integration in Software Testing

In continuous integration, the work done by team members is combined often, at least by daily by everyone into the system they are developing. When the integration is done it should be tested automatically so the possible errors would be noted as soon as possible. As already mentioned, many teams have noticed that this way integration errors are reduced, and software development is faster. (Fowler, 2006)

In CI, the developer commits code to the version control repository while the CI server checks for changes with small intervals from the repository. After the CI server detects the changes in a version control repository the CI server retrieves new code and starts to build a script. When the build script gets everything done the CI server reports the result to specified project members by email or other decided way. After everything is done the CI server goes back to checking for changes in version control repositories

until the developer commits new code. (Duvall, Glover, and Matyas, 2017) The working CI system is presented in picture 1.

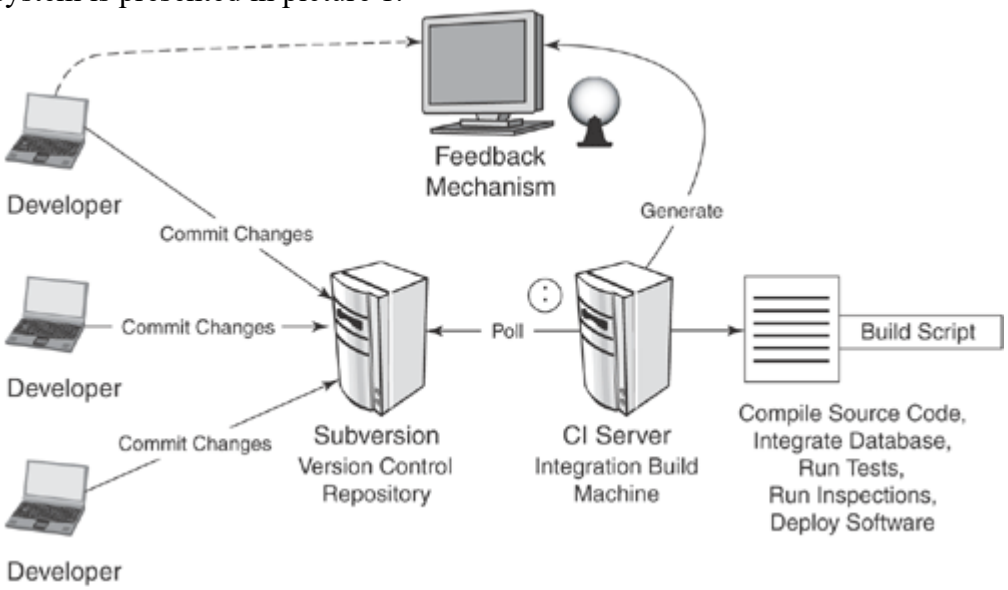

Figure 1. Components of a continuous integration system (Duvall, Glover, and Matyas, 2017)

According to Duvall, Glover, and Matyas (2017):

"At a high level, the value of CI is to:

- Reduce risks
- Reduce repetitive manual processes
- Generate deployable software at any time and any place
- Enable better project visibility
- Establish greater confidence in the software product from the development team"

According to Demarey (2013), there are ten rules on how to apply continuous integration:

"

1. Check-in regularly to mainline
2. Have an automated test suite
3. Always run tests locally before committing
4. Don't commit broken code
5. Don't check-in on a broken build
6. Fix broken build immediately
7. Time-box fixing before reverting
8. Never go home on a broken build
9. Don't comment out failing tests
10. Keep your build fast"

3 Continuous Integration in System-on-Chip Design

The next chapter compares continuous integration in hardware to continuous integration to the software. The chapter defines problems for applying continuous integration from the software world to the hardware world. After the problems have been defined there is a discussion of how the problems could be solved. References speak of hardware overall, but in this thesis, the hardware is seen as the same as SoC.

3.1 CI in Hardware Compared to CI in Software

In software, there may be only one product that is developed continuously until the end of its support, while in hardware development, the idea of the product is to have only one and final release (Cerisier & Rivier, n.d.). Figure 2 shows how hardware development starts from the beginning always when a new product is developed and how the old product is not maintained any more (Yoshioka, Marte, Fontana, Oliveira, & Yano, 2013). While Figure 3 shows how software development starts cycling around because of the maintenance period.

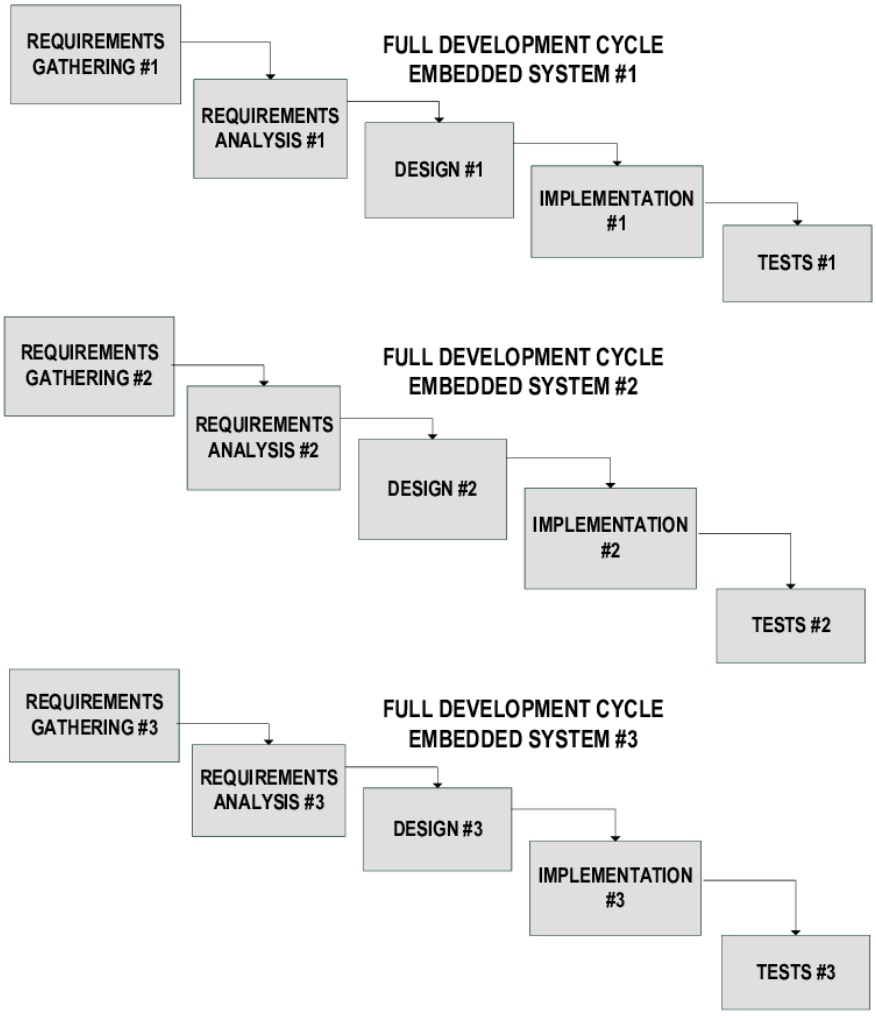

Figure 2. The full development cycle of three different hardware products (Yoshioka, Marte, Fontana, Oliveira, & Yano, 2013)

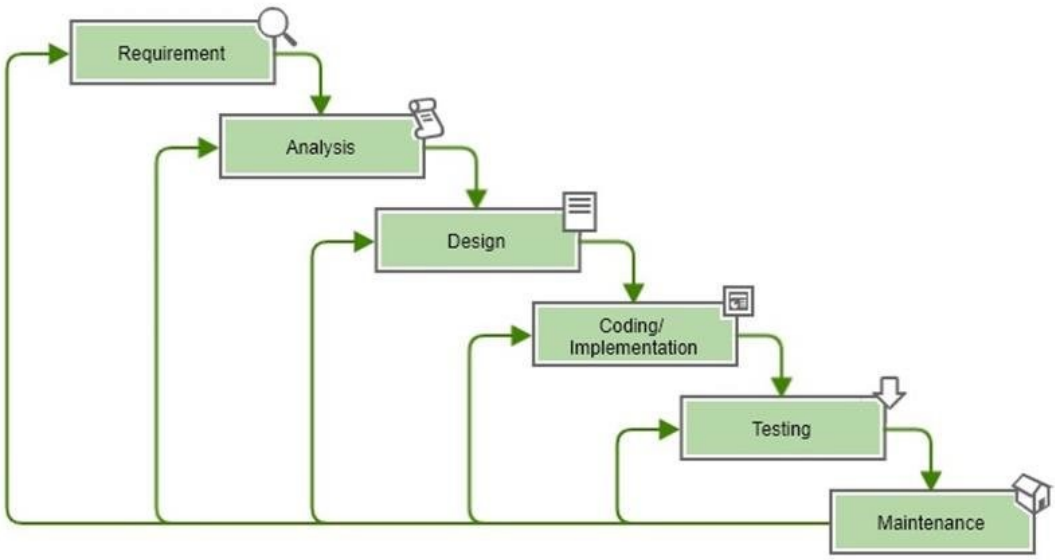

Figure 3. The software development cycle of a product (Widiaty, Riza, Ana, Abdullah, & Mubaroq, 2019)

Naturally, in hardware, some functional blocks are developed, integrated, and updated during the project. Functional blocks are parts that are possible to apply some continuous integration rules (Cerisier & Rivier, n.d.).

The ten rules presented in the last chapter can be adjusted to hardware with the next pieces of advice:

1. Check-in regularly to the mainline

In Register-transfer level development engineers need to design their RTL carefully so it will meet all the requirements and restrictions. If the intellectual property is from an external provider, it might take days to understand the intellectual property and make it fully functional. Because of that check-ins are hard to deliver several times per day by the designer while keeping the integration functional. This is no problem at the end of the day if check-in is done as soon as there is an implementation of new functionality. (Cerisier & Rivier, n.d.)

2. Have an automated test suite

In hardware, automated self-checking tests are implemented by verification teams and not by hardware development teams. Because tests and product development are done by different teams, it might take some time before the newly released function gets related tests implemented for it. (Cerisier & Rivier, n.d.)

3. Always run tests locally before committing

It is supposed to be possible for designer run tests if they are available, but as mentioned in point two, it might be impossible because tests are not yet made available by the verification team. A feature that has been checked-in may not be tested at all and then the

feature is considered to be a primary release that verification team can use for testing the design. (Cerisier & Rivier, n.d.)

4. Don’t commit broken code to the mainline

Occasionally hardware designers have a good reason to commit broken code to mainline. These cases are, for example, when releasing structural RTL that does not work for the design for test, supplier technical assistance, or backend-end -engineers so they could start a structural analysis of the code. If it is not possible to avoid the situation where broken code is committed to the mainline there should be some workarounds implemented which keep the status on “Green”. Even though the workarounds mark the status “Green” there should be some progress indicators that inform about the existence of workaround and reports a failed test suite until RTL is fixed and no workaround is needed anymore. (Cerisier & Rivier, n.d.)

5. Don’t check-in on a broken build

This point goes very close to point four because the same idea applies. More matters should be noticed because they could interrupt checking the functionality of the new RTL function developed, for example, tests are not available like mentioned in point two, the simulation takes too much time or it is not known how to run test suit made by the design team. (Cerisier & Rivier, n.d.)

6. Fix broken build immediately

Verifiers and designers are in different teams and it can cause problems with fast fixes of the broken build because the problem might be under another team. Another team might not have the time or resources to fix it immediately because they are prioritizing some other work at the time. (Cerisier & Rivier, n.d.)

7. Time-box fixing before reverting

In software, developers spend time for debugging usually 10 or 20 minutes of a broken build. In hardware, it might take 30 minutes or even 6 hours to have a single simulation and the debug time can be days before having an RTL solution for it. (Cerisier & Rivier, n.d.)

8. Never go home on a broken build

Besides, because of the debug times mentioned in point seven, it is not possible to keep working until the build is fixed. The way in hardware is to prefer working problems on a local branch and keep the main branch working. (Cerisier & Rivier, n.d.)

9. Don’t comment out failing tests

In some cases, hardware teams just comment out a test that does not pass so they would give false “green” but this is not preferable because it hides parts that do not function properly. (Cerisier & Rivier, n.d.)

10. Keep your build fast

The software build is recommended to keep 10 minutes to run (Fowler, 2006). In case it takes longer it might be possible to use pipeline stages to balance the load to multiple stages. However, on the system-on-chip level, it may take an hour or even days depending on how complex the system-on-chip is. Because of those problems, this point should be applied by using more stages in the pipeline so problems would be detected as fast as possible. (Cerisier & Rivier, n.d.)

3.2 Problems of CI in System-on-Chip Design

As addressed earlier, there are problems for applying continuous integration to the system-on-chip design directly in the way it is used in software development. To summarize those problems defined by Cerisier and Rivier (n.d.) are the following:

- RTL development is needed to accomplish very carefully so it will meet all the requirements and restrictions.
- Tests and product development are done by different teams.
- Forced broken code commits to the mainline.
- Tests are not available.
- The simulation takes too much time.
- It is not known how to run test suits.
- Long debugging time.
- Commenting out failing tests.
- No possibility for fast fixes.

It is possible to see the dependencies between these problems. In this thesis, those dependencies are given by naming five main problems and several subproblems. When the main problem is solved the subproblems will be solved even though some main problems might not be possible to fix. Some of the main problems might not have subproblems and some of the main problems might, additionally, be under other main problems and some subproblems might be part of many main problems.

3.2.1 RTL development is needed to do very carefully so it will meet all the requirements and restrictions.

As Cerisier and Rivier (n.d.) state RTL development must be done very carefully because requirements and restrictions it might be that this problem cannot be solved. Even though it might be unsolvable designers should consider ways they could make it faster so that it would be possible to make multiple check-ins per day.

3.2.2 Tests and product development are done by different teams

The problem of different teams is connected to subproblems such as:

- It is not known how to run test suits.

- No possibility for fast fixes.
- Tests are not available.

And these problems could be solved by making only one team that would be responsible for product development and testing. If it is not possible to combine these teams successfully, the solution could be making more co-operative teams that would prioritize tasks made by specific teams while offering more time for the tasks. If neither of these would work, some other ways should be thought which makes problems consisted of different teams disappear.

3.2.3 The simulation takes too much time

Time dependency is an issue that affects debugging time and functionality of committed code. Simulation time could be solved with faster servers making faster simulations but because it is not the point of continuous integration that might not be the best solution. As Cerisier and Rivier (n.d.) stated the build should have multiple stages so problems would be noticed as fast as possible. The solution could also be a way to cut simulation to pieces so that only the important pieces would run.

3.2.4 Commenting out failing tests.

The problem where hardware teams just comment out the failing test so the build would pass is simply corrected by way Cerisier and Rivier (n.d.) gave, do not comment out failing tests and if it is necessary to have nonfunctional code committed then make an indicator that shows that there are problems that should be solved instead of false “green”.

3.3 Usage of Continuous Integration in System-on-Chip

Jenkins is a tool for creating an automation server that can be used in continuous integration servers. To get the most out of Jenkins server it is possible to extend Jenkins with multiple add-ons so the tool would be most suitable for specific usage. (Jenkins, n.d.)

Cerisier and Rivier (n.d.) presented a customer use case using the Jenkins server, with a few plugins and python scripts for reporting, for continuous integration. In the use case, there are tests separated into three different cases which will be referred to as sanity-, daily-, and weekly tests.

While talking about this case, sanity tests are lightweight tests that take about 20 minutes to run and test basic functionalities in blocks. Sanity tests are run for every commit. (Cerisier & Rivier, n.d.) With sanity testing problems caused by too long test times are reduced because sanity tests are fast, and the developer gets report if commit breaks some basic functionality.

Daily tests that are run can take hours and they are run every night (Cerisier & Rivier, n.d.). The weekly tests are like the name suggests ran every week. The weekly test can

take time for a couple of days because it checks all the functionality in the chip. (Cerisier & Rivier, n.d.)

4 Discussion

Implementation of continuous integration can be done with some tools, for example, Jenkins but it is not implemented in the same way as in the software because development differs between hardware and software. The most problems that occur when applied continuous integration directly to the system-on-ship design are possible to solve now if we can flex a little from the strict line. If we notice that it might not be possible to apply directly continuous integration to the system-on-chip design, we could only use the suitable parts from continuous integration and try to make solutions to unsuitable parts if possible without breaking the whole system.

The implementation can be done somehow if followed by the guidance of Cerisier and Rivier (n.d.) but it is not the perfect solution. To improve that solution more research should be done about the implementation of continuous integration to the system-on-chip design.

As presented in this thesis continuous integration could make development faster and open more rapid movements as it has done in software development. The limitation of this thesis is that there is not much relevant research data on continuous integration in system-on-chip design. Therefore the topic should be studied more.

One possible research subject in the following researches that could be studied is the team structures in system-on-chip design: would it be possible to combine different teams used in development to one team. In addition to previous, any other problem that affects the usage of continuous integration in system-on-chip design could be researched and besides, the overall usage could be researched.

5 Conclusion

At first, this thesis defined the concepts of RTL, system-on-chip, field-programmable gate array, and application-specified integrated circuit. After defining the concepts, this thesis moved on to explaining what continuous integration generally is and how it is used in software development.

The continuous integration guided us to the main topic of this thesis: continuous integration in the system-on-chip design. Concerning continuous integration in the system-on-chip design, there was first an introduction to the differences between continuous integration in hardware and software. After that, the thesis presented the main problems of continuous integration usage on the system-on-chip design compared to the software.

The research question of this thesis was can continuous integration be used in the system-on-chip design. As this thesis essentially has pointed out it is mainly possible to

apply some continuous integration methods in the system-on-chip design to get similar results as software development gets by applying them. Continuous integration methods cannot be applied directly, and some modification needs to be done because software development works differently than hardware design.

There is not much research done that would give information about applying continuous integration on the system-on-chip design and for fully applying continuous integration to the system-on-chip design it should be researched more.

References

Cerisier, F., & Rivier, C. (n.d.). *Applying Continuous Integration to Hardware Design and Verification*. https://www.design-reuse.com/articles/41235/applying-continuous-integration-to-hardware-design-and-verification.html, Retrieved 20.4.2020.

Chu, P. (2006). *RTL hardware design using VHDL: coding for efficiency, portability, and scalability.* https://doi.org/10.1002/0471786411

Churiwala, S., & Garg, S. (2011). *Principles of VLSI RTL Design A Practical Guide* (1st ed. 2011.). https://doi.org/10.1007/978-1-4419-9296-3

Demarey, C. (2013) *What is Continuous integration?* http://chercheurs.lille.inria.fr/~demarey/Main/ContinuousIntegration, Retrieved 20.4.2020.

Duvall, P., Glover, A., & Matyas, S. (2007). *Continuous integration: improving software quality and reducing risk*. Place of publication not identified: Addison Wesley.

Flynn, M., & Luk, W. (2011). *Computer System Design System-on-Chip.* Hoboken: Wiley.

Fowler, M. (2006) *Continuous Integration.* www.martinfowler.com/articles/continuousIntegration.html, Retrieved 20.4.2020.

Hsu, P., & Liu, Y. (2014). *Buffer Design and Assignment for Structured ASIC.* Journal of Information Science and Engineering, 30(1), 107–124.

Jenkins (n.d.). https://jenkins.io, Retrieved 20.4.2020.

Monmasson, E., & Cirstea, M. (2007). *FPGA Design Methodology for Industrial Control Systems-A Review.* IEEE Transactions on Industrial Electronics, 54(4), 1824–1842. https://doi.org/10.1109/TIE.2007.898281

Olausson, M., & Ehn, J. (2015). *Continuous Delivery with Visual Studio ALM 2015* (1st ed. 2015.). https://doi.org/10.1007/978-1-4842-1272-1

Widiaty, I., Riza, L., Ana, A., Abdullah, M., & Mubaroq, S. (2019). *Web-Based Digital Learning Application of Iconic Batik in Batik Learning at Vocational High School.* Journal of Engineering Science and Technology, 14(5), 2475-2484.

Yoshioka, L., Marte, C., Fontana, C., Oliveira, M., & Yano, E. (2013). *Telematic Device Development Based on Framework for Embedded Systems.*

Zhang, P., (2010). *Programmable-logic and application-specific integrated circuits (PLASIC).* Advanced Industrial Control Technology, 215–253, https://doi.org/10.1016/B978-1-4377-7807-6.10006-3